Eric R. Schellhammer und Benjamin B. Hargarten

Werkzeuge zur Entwicklung von Statecharts für interaktive Systeme

Diplomarbeit im Studiengang Mathematik
mit Studienrichtung Informatik

schell@informatik.uni-hannover.de
hargarten@informatik.uni-hannover.de

Universität Hannover – Institut für Informatik
Fachgebiet Datenbanken und Informationssysteme
Prüfer: Prof. Dr. U. Lipeck Zweitprüfer: Prof. Dr. R. Parchmann

5. Oktober 2000

Hiermit versichern wir, dass wir diese Arbeit selbständig verfasst haben, und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.

Hannover, 5. Oktober 2000; Eric R. Schellhammer

Hannover, 5. Oktober 2000; Benjamin B. Hargarten

Now hear me speak
with a prophetic spirit.
(Shakespeare, King John)

Inhaltsverzeichnis

I Diskussion der Statecharts vom theoretischen Standpunkt 15

1 Definition von Statecharts 17
1.1 Grundlegende Definitionen . . . 17
1.1.1 Zustände . . . 18
1.1.2 Übergänge (Transitionen) . . . 19
1.2 Formale Definition . . . 19
1.2.1 Statechart . . . 20
1.2.2 Die Menge der Zustände Z . . . 20
1.2.3 Die Unterzustandsfunktion σ . . . 20
1.2.4 Die Partitionsfunktion π . . . 22
1.2.5 Die Defaulteinstiegsfunktion d . . . 24
1.2.6 Die Historyfunktion h und die Menge der Historyeinstiege H 24
1.2.7 Die Menge der Ereignisse E . . . 24
1.2.8 Die Menge der Aktionen A . . . 25
1.2.9 Die Menge der Kanten K . . . 25
1.2.10 Die Sprache der Bedingungen L_B . . . 26
1.3 Grafische Definition . . . 27
1.3.1 Zustände . . . 27
1.3.2 Übergänge . . . 28
1.3.3 Defaulteinstieg . . . 30
1.3.4 Historyeinstieg . . . 30
1.3.5 Konnektoren . . . 31
1.4 Temporale Auswertung . . . 31
1.4.1 Der Zustand eines Statecharts . . . 31
1.4.2 Der Zeitschritt . . . 32
1.5 Abweichungen . . . 35

2 Varianten von Statecharts in der Literatur 37
2.1 Einleitung . . . 37
2.2 Zustände . . . 38
2.2.1 Grundformen der atomaren Zustände . . . 38
2.2.2 Zusammengesetzte Zustände . . . 39
2.2.3 Parametrisierte Zustände . . . 39
2.2.4 Denkbare Erweiterungen des Zustandsbegriffs . . . 40

2.3 Konnektoren . . . 42
2.3.1 Grundformen der Konnektoren . . . 42
2.3.2 Der Historykonnektor . . . 45
2.4 Verbindungen . . . 46
2.4.1 Zusammengesetzte Übergänge . . . 47
2.4.2 Determinismus und Prioritäten . . . 49
2.4.3 Denkbare Erweiterungen des Übergangsbegriffs . . . 50
2.5 Ereignisse . . . 51
2.5.1 Sichtbarkeitsdauer . . . 51
2.5.2 Die Rahmenbedingung . . . 52
2.6 Bedingungen . . . 53
2.7 Aktionen . . . 54
2.8 Der Zeitbegriff . . . 54
2.8.1 Kontinuierlich, synchron oder asynchron . . . 54
2.8.2 Delays und Time-Outs . . . 55

3 Werkzeuge für Statecharts 57
3.1 Grafische Editoren . . . 57
3.1.1 Der Statechart-Editor . . . 57
3.1.2 Der Dialog-Editor . . . 66
3.1.3 Der Programmstruktur-Editor . . . 69
3.2 Tester . . . 71
3.2.1 Der Verifikator . . . 71
3.2.2 Der Simulator . . . 77
3.3 Automatismen . . . 83
3.3.1 Der Optimierer . . . 83
3.3.2 Der Quelltext-Generator . . . 86

II Grundlagen von *SCed* 91

4 Die implementierte Fassung 93
4.1 Einleitung . . . 93
4.2 Die Grundelemente . . . 93
4.2.1 Zustände . . . 93
4.2.2 Konnektoren . . . 97
4.2.3 Ereignisse, Aktionen und Bedingungen . . . 97
4.2.4 Die Variablen . . . 101
4.2.5 Die Zeit . . . 102
4.3 Die Dialoge . . . 102

5 Benutzerhandbuch 105
5.1 Installation . . . 105
5.2 Überblick über das Programm *SCed* . . . 105
5.3 Der Inspektor . . . 106
5.3.1 Inspektor für einen Zustand . . . 107
5.3.2 Inspektor für einen Pfeil . . . 108

5.3.3 Inspektor für einen Text . . . 109
5.3.4 Weitere Elemente im Inspektor . . . 109
5.3.5 Menüs . . . 110
5.4 Der grafische Editor . . . 112
5.4.1 Zeichenfläche . . . 112
5.4.2 Werkzeugleiste . . . 115
5.4.3 Die Eigenschaftsleiste . . . 118
5.5 Der Dialog-Editor . . . 118
5.5.1 Untypische Dialogkomponenten . . . 119
5.5.2 Kommunikation über Variablen . . . 120
5.6 Bedingungen . . . 120
5.7 Ereignisse . . . 121
5.8 Aktionen . . . 122
5.9 Der Simulator . . . 123
5.10 Undo/Redo . . . 125
5.11 Quelltextexport . . . 126
5.11.1 Exportieren des Statecharts . . . 126
5.11.2 Methoden der Quelltextsegmente . . . 126
5.12 Erweitern des Programms um neue Dialogkomponenten . . . 127
5.13 Der Verifikator . . . 127
5.13.1 Überblick . . . 127
5.13.2 Die einzelnen Tests . . . 128

6 Funktionsweise der Module 135
6.1 Überblick . . . 135
6.2 Die Datenstruktur des Statecharts . . . 135
6.2.1 Überblick . . . 135
6.2.2 Das Logische Statechart-Modell . . . 137
6.2.3 Das Grafische Statechart-Modell . . . 138
6.2.4 Das Simulations- und Quelltextmodell . . . 141
6.3 Das logische Dialogmodell . . . 142
6.4 Der Editor . . . 143
6.5 Der Simulator . . . 144
6.5.1 Verwaltete Daten . . . 144
6.5.2 Der Simulationsablauf . . . 145
6.5.3 Aktivieren in der Zustandshierarchie . . . 147
6.6 Der Verifikator . . . 149
6.7 Quelltextexport . . . 149
6.8 Das Kommandointerface – Laden und Speichern . . . 150
6.9 Ausblick . . . 150

III Anhang: Beispiel und Programmdokumentation 153

A Beispiele 155
A.1 Die Teemaschine . . . 155
A.1.1 Beschreibung des Systems . . . 155

A.1.2 Elemente des Dialogs . . . 155
A.1.3 Umsetzung in ein Statechart . . . 157
A.1.4 Unterschiede des Modells zur echten Teemaschine . . . 164
A.2 Ein einfacher Anrufbeantworter . . . 164
A.2.1 Beschreibung des Systems . . . 164
A.2.2 Elemente des Dialogs . . . 165
A.2.3 Umsetzung in ein Statechart . . . 166
A.2.4 Möglichkeiten der Verfeinerung . . . 169

B Pflichtenheft 171
B.1 Zielbestimmung . . . 171
B.1.1 Muss-Kriterien . . . 171
B.1.2 Kann-Kriterien . . . 172
B.1.3 Abgrenzungskriterien . . . 172
B.2 Einsatz . . . 172
B.2.1 Anwendungsbereiche . . . 172
B.2.2 Zielgruppe . . . 172
B.3 Umgebung . . . 173
B.3.1 Software . . . 173
B.3.2 Hardware . . . 173

C Grammatik des Datenformats 175

D Programmdokumentation 179
D.1 Package score . . . 179
D.1.1 Klassenhierarchie . . . 179
D.1.2 Klasse start . . . 179
D.2 Package score.AbstractModell . . . 179
D.2.1 Klassenhierarchie . . . 179
D.2.2 Klasse AbstractModell . . . 179
D.2.3 Klasse MarkableTyp . . . 181
D.3 Package score.Aktionen . . . 182
D.3.1 Klassenhierarchie . . . 182
D.3.2 Interface Actor . . . 182
D.3.3 Interface DialogAktion . . . 183
D.3.4 Interface EventHolder . . . 183
D.3.5 Interface HistoryAktion . . . 184
D.3.6 Klasse Aktion . . . 184
D.3.7 Klasse AktionDialog . . . 185
D.3.8 Klasse AktionEdit . . . 186
D.3.9 Klasse AktionProperty . . . 186
D.3.10 Klasse AktionPropertyChange . . . 187
D.3.11 Klasse AktKomplex . . . 188
D.3.12 Klasse CalledAction . . . 189
D.3.13 Klasse ClearHistoryAktion . . . 189
D.3.14 Klasse DeepClearHistoryAktion . . . 190
D.3.15 Klasse ExportAktion . . . 190

D.3.16 Klasse IncVarAktion 191
D.3.17 Klasse JavaAktion 192
D.3.18 Klasse SetDialogOnAktion 192
D.3.19 Klasse SetDialogVisibleAktion 193
D.3.20 Klasse SetVarAktion 194
D.3.21 Klasse ThrowAktion 195
D.4 Package score.Bedingungen 196
D.4.1 Klassenhierarchie 196
D.4.2 Interface ConditionAskable 196
D.4.3 Klasse BedAND 196
D.4.4 Klasse BedAtomar 197
D.4.5 Klasse Bedingung 197
D.4.6 Klasse BedKomplex 198
D.4.7 Klasse BedNOT 199
D.4.8 Klasse BedOR 200
D.4.9 Klasse CompVarBed 200
D.4.10 Klasse ExportBedingung 201
D.4.11 Klasse HasHistoryBed 201
D.4.12 Klasse InStateBed 202
D.4.13 Klasse JavaBed 202
D.5 Package score.control 203
D.5.1 Klassenhierarchie 203
D.5.2 Interface Logger 203
D.5.3 Klasse AbstractParser 204
D.5.4 Klasse BedParser 204
D.5.5 Klasse Commander 205
D.5.6 Klasse CommandPort 207
D.5.7 Klasse CommandWindow 208
D.5.8 Klasse Inspector 208
D.5.9 Klasse Interpreter 208
D.5.10 Klasse MainWindow 209
D.5.11 Klasse Parser 210
D.5.12 Klasse PropertyView 211
D.5.13 Klasse Scanner 212
D.5.14 Klasse SCInterpreter 212
D.5.15 Klasse Token 213
D.6 Package score.DEditor 213
D.6.1 Klassenhierarchie 213
D.6.2 Klasse DialogEditor 213
D.6.3 Klasse DlgKompSelektor 214
D.7 Package score.doclet 214
D.7.1 Klassenhierarchie 214
D.7.2 Klasse LaTeXDoclet 215
D.8 Package score.Ereignisse 216
D.8.1 Klassenhierarchie 216
D.8.2 Interface EventAskable 216
D.8.3 Klasse Ereignis 216

D.8.4 Klasse EreignisEdit 217
D.8.5 Klasse EventDialog 217
D.8.6 Klasse InlineEventDialog 218
D.9 Package score.GrafSCM 218
D.9.1 Klassenhierarchie 218
D.9.2 Error InvalidConstructionError 219
D.9.3 Klasse GPAbschnitt 219
D.9.4 Klasse GrafAND 221
D.9.5 Klasse GrafHook 222
D.9.6 Klasse GrafKonnektor 225
D.9.7 Klasse GrafModellSicht 226
D.9.8 Klasse GrafModellSicht.visHook 231
D.9.9 Klasse GrafModellSicht.visText 231
D.9.10 Klasse GrafPfeil 231
D.9.11 Klasse GrafPunkt 234
D.9.12 Klasse GrafSCModell 239
D.9.13 Klasse GrafText 240
D.9.14 Klasse GrafTopLevelZust 241
D.9.15 Klasse GrafTyp 242
D.9.16 Klasse GrafZustand 243
D.9.17 Klasse Partitioner 247
D.9.18 Klasse SuperList 249
D.9.19 Klasse Unterpunkt 249
D.10 Package score.LogDM 250
D.10.1 Klassenhierarchie 250
D.10.2 Interface DialogCompManager 250
D.10.3 Interface DialogCompServer 250
D.10.4 Klasse DlgButton 251
D.10.5 Klasse DlgCheckbox 251
D.10.6 Klasse DlgKomponente 252
D.10.7 Klasse DlgPopup 254
D.10.8 Klasse DlgRadiobutton 255
D.10.9 Klasse DlgText 255
D.10.10 Klasse DlgTimer 256
D.10.11 Klasse LogDModell 257
D.11 Package score.LogSCM 258
D.11.1 Klassenhierarchie 258
D.11.2 Klasse KonDefault 259
D.11.3 Klasse KonHistory 259
D.11.4 Klasse Konnektor 259
D.11.5 Klasse KonUniversal 260
D.11.6 Klasse LogComponent 260
D.11.7 Klasse LogPfeil 260
D.11.8 Klasse LogPfeil_1_1 262
D.11.9 Klasse LogPfeil_1_N 263
D.11.10 Klasse LogPfeil_N_1 264
D.11.11 Klasse LogPunkt 265

D.11.12 Klasse LogSCModell 267
D.11.13 Klasse ZustAND 268
D.11.14 Klasse ZustandAtom 268
D.11.15 Klasse ZustKomplex 269
D.11.16 Klasse ZustOR 270
D.12 Quelltext-Export 270
D.12.1 Klassenhierarchie 270
D.12.2 Klasse ExportVariable 271
D.12.3 Klasse StateChart 271
D.12.4 Klasse Timer 272
D.13 Package score.SCEditor 273
D.13.1 Klassenhierarchie 273
D.13.2 Klasse BufferedCanvas 273
D.13.3 Klasse DrawableRubber 274
D.13.4 Klasse Painter 275
D.13.5 Klasse RubberBand 275
D.13.6 Klasse RubberCircle 276
D.13.7 Klasse RubberFilledCircle 276
D.13.8 Klasse RubberOval 276
D.13.9 Klasse RubberRectangle 277
D.13.10 Klasse RubberRoundRect 277
D.13.11 Klasse RubberSet 278
D.13.12 Klasse RubberText 279
D.13.13 Klasse SCCanvas 279
D.13.14 Klasse SCEditor 280
D.13.15 Klasse ScrollCanvas 283
D.13.16 Klasse TranslationWindow 283
D.14 Package score.SimSCM 284
D.14.1 Klassenhierarchie 284
D.14.2 Exception NoTargetStateException 284
D.14.3 Klasse SimAND 284
D.14.4 Klasse SimBasic 285
D.14.5 Klasse SimConn 287
D.14.6 Klasse SimHist 287
D.14.7 Klasse SimOR 288
D.14.8 Klasse SimSCModell 289
D.15 Package score.Simulator 290
D.15.1 Klassenhierarchie 290
D.15.2 Interface VarServer 290
D.15.3 Klasse SCRun 290
D.15.4 Klasse Simulation 292
D.15.5 Klasse SimVariable 293
D.15.6 Klasse SimWindow 293
D.15.7 Klasse Variable 294
D.16 Package score.util 295
D.16.1 Klassenhierarchie 295
D.16.2 Interface ErrorPrinter 296

D.16.3 Interface Observer . . . 296
D.16.4 Interface UndoManager . . . 296
D.16.5 Klasse AnimatedImage . . . 296
D.16.6 Klasse BlackLabel . . . 297
D.16.7 Klasse Compare . . . 297
D.16.8 Klasse Dimension . . . 298
D.16.9 Klasse Dimension.Double . . . 298
D.16.10 Klasse EasyFileWriter . . . 299
D.16.11 Klasse EasyStringWriter . . . 300
D.16.12 Klasse FlatButton . . . 300
D.16.13 Klasse FlatRadioButton . . . 300
D.16.14 Klasse Intersect . . . 301
D.16.15 Klasse LaTeXString . . . 301
D.16.16 Klasse MetaPostGraphics . . . 301
D.16.17 Klasse NumField . . . 303
D.16.18 Klasse Observable . . . 303
D.16.19 Klasse Pair . . . 304
D.16.20 Klasse Scale . . . 304
D.16.21 Klasse StringCheck . . . 305
D.16.22 Klasse TextEditField . . . 305
D.16.23 Klasse Triplet . . . 306
D.16.24 Klasse UndoStack . . . 306
D.16.25 Klasse VecArithmetik . . . 307
D.16.26 Klasse ZoomGraphics . . . 307
D.17 Package score.Verifikator . . . 309
D.17.1 Klassenhierarchie . . . 309
D.17.2 Klasse Verifikator . . . 309
D.17.3 Klasse VProperties . . . 310
D.17.4 Klasse VPropertyWindow . . . 310
D.17.5 Klasse VWindow . . . 311
D.17.6 Klasse VWindow.VerifMesg . . . 311

E Aufteilung dieser Arbeit **313**

Abbildungsverzeichnis **317**

Dimidium facti, qui coepit, habet.[1]
(Horaz, „Episteln“ 1, 2, 40)

Einleitung

Überblick über die Arbeit

Diese Arbeit soll einen Einblick in die Modellierung von Benutzerschnittstellen mit der Hilfe von Statecharts geben. Dafür werden wir Statecharts zunächst als Erweiterung von Zustands-Übergangs-Diagrammen vorstellen und im Anschluss formal definieren, was wir unter einem Statechart verstehen wollen und welche Semantik dahintersteht. Dabei werden wir hauptsächlich von der Fassung ausgehen, in der sie David Harel 1987 erstmals vorgestellt hat. Danach wird ein Überblick über einige unterschiedliche Varianten und Erweiterungen gegeben, die Statecharts in der Literatur erfahren haben.

In Kapitel 3 werden dann einige Werkzeuge diskutiert, die für die Arbeit mit Statecharts denkbar sind. Hierbei wird sowohl auf die Frage eingegangen, welche Möglichkeiten diese Werkzeuge bieten sollten, als auch Beschränkungen solcher Programme aufgezeigt.

Im zweiten Teil soll der Blick genauer auf das Programm *SCed* gelenkt werden, das einige der vorher besprochenen Werkzeuge umsetzt. Zunächst wird beschrieben, welche Form der Statecharts dem Programm zugrunde liegt, und diese Auswahl wird begründet. In Kapitel 5 folgt das Benutzerhandbuch, das die Funktionen von *SCed* vorstellt und das Arbeiten mit diesen erklärt. Kapitel 6 schließlich wird auf dieser Grundlage erläutern, wie die einzelnen Teile von *SCed* zusammenarbeiten.

Zwei Beispiele, das Pflichtenheft für das Programmpaket, eine Beschreibung, wie diese Arbeit unter den Autoren aufgeteilt wurde, sowie die genaue Programmdokumentation befinden sich im Anhang.

Wissenschaftliche Einordnung

Statecharts werden dazu benutzt, das Verhalten von interaktiven Systemen zu beschreiben und zu modellieren. David Harel stellte sie in [Har87b] als Erweiterung von Zustands-Übergangs-Diagrammen vor, indem er die Konzepte der Tiefe und Parallelität hinzufügte. Dadurch sind Statecharts den für endliche Automaten üblichen Zustands-Übergangs-Diagrammen deutlich überlegen.

[1] Frisch gewagt ist halb gewonnen. (wörtlich: Die Hälfte des Ganzen hat, wer angefangen hat.)

Horrocks unternimmt diesen Vergleich[2] mit einem relativ einfachen Modell eines CD-Spielers und kommt zu dem Schluss, dass Zustands-Übergangs-Diagramme eine unübersichtlich hohe Anzahl von Zuständen (und Übergängen) brauchen, wohingegen eine äquivalente Formulierung in Statecharts sehr leicht zu verstehen ist. In einem weiteren Beispiel (ein Texteditor mit verschiedenen Möglichkeiten, Text zu formatieren) gibt er konkrete Zahlen an: Den 13 Zuständen und 16 Pfeilen des Statecharts stünden hier 96 Zustände und 7776 Pfeile im entsprechenden Zustands-Übergangs-Diagramm entgegen.[3]

Es sei hier noch erwähnt, dass Horrocks auch den Vergleich zur natürlichsprachlichen Beschreibung der Verhaltensweise eines Systems zieht. Dieser Form der Beschreibung kann man zwar zugute halten, dass sie auch von Laien verstanden werden kann – doch sind insbesondere der Umfang und die Tendenz zur Unvollständigkeit Argumente, die Statecharts als die bessere Alternative bekräftigen.[4]

Durch diese deutlichen Vorteile ist es nicht verwunderlich, dass Statecharts zunehmend Beachtung finden. Es wurden zahlreiche Arbeiten veröffentlicht, die sich mit dem Thema beschäftigen und verschiedene Varianten vorschlagen, sowohl in Umfang und Aussehen der grafischen Darstellung als auch in der Bedeutung der einzelnen Elemente. So hat Harel selbst einige von seiner ersten Fassung abweichende Versionen beschrieben, und auch in die Unified Modelling Language (UML) fanden Statecharts Eingang. Darüber hinaus wurden auch Werkzeuge entwickelt, die das Arbeiten mit Statecharts erleichtern. Von diesen sollte insbesondere das Programm Statemate erwähnt werden, das unter anderem von David Harel selbst mitentwickelt wurde.

Für genauere Details dieser Varianten sei hier jedoch zunächst auf Kapitel 2 verwiesen.

[2] vgl. [Hor99a], S. 37-55
[3] vgl. [Hor99a], S. 71
[4] vgl. [Hor99a], S. 54 ff.

Teil I

Diskussion der Statecharts vom theoretischen Standpunkt

Let there be light.
(Genesis (1, 3))

Kapitel 1

Definition von Statecharts

Im Folgenden wollen wir eine Definition von Statecharts als Grundlage für die nachfolgende Diskussion erstellen. Wir beziehen uns dabei im Wesentlichen auf [Har88]. Da in der Literatur die Definition in eher informeller Weise durchgeführt wird, geben wir hier keinen direkten Bezug auf Literatur anderer Autoren an.

Wir beginnen unsere Definition mit den grundlegenden Elementen des Statecharts. Danach definieren wir formal den Aufbau der Statecharts. In Abschnitt 1.3 verbinden wir dann diesen formalen Aufbau mit einer grafischen Notation. Es folgen in Abschnitt 1.4 einige Definition, die das Verhalten des Statecharts in einem Ablauf wiedergeben.

1.1 Grundlegende Definitionen

Statecharts bestehen aus den folgenden Elementen:

- Zustände
- Übergänge
- Bedingungen
- Ereignisse
- Aktionen

Vorbemerkungen Hilfreich ist es, wenn man die Definition von endlichen Automaten betrachtet und Statecharts als solche, erweitert um Tiefe und Nebenläufigkeit, ansieht. Weiter ist zu beachten, dass wir für die Abarbeitung der Statecharts eine diskrete Zeit $t \in \mathbb{N}$ benutzen. Hier gibt es in der Literatur auch andere Konzepte.

Um sich mit den Elementen des Statcharts vertraut zu machen, betrachte man zum Beispiel Abbildung A.3 auf Seite 158, in der alle wesentlichen Elemente der Statecharts benutzt werden.

1.1.1 Zustände

Ein Zustand ist ein abstraktes Gebilde, dessen Aufgabe in dem Wissen liegt, ob er aktiviert oder deaktiviert ist. Zustände haben eine Identifikation und sind dadurch unterscheidbar. Desweiteren gibt es die Ereignisse

- Zustand betreten
- Zustand verlassen

Man kann in Bedingungen prüfen, ob bestimmte Zustände:

- aktiv sind
- inaktiv sind

Ein Zustand kann weiterhin von folgendem Typ sein:

- atomar
- zusammengesetzt (Typ OR)
- zusammengesetzt (Typ AND)

Diese Unterscheidung entsteht dadurch, dass Zustände in anderen Zuständen, also hierarchisch untergeordnet liegen können.

Atomare Zustände

Atomare Zustände haben keine inneren Zustände, können jedoch zu einem Oberzustand gehören.

Zusammengesetzte Zustände (Typ OR)

Dieser Zustand besitzt eine Menge von Unterzuständen. Ist der Zustand aktiviert, so auch genau ein direkter Unterzustand. Wir verwenden für diesen Zustand, wie in der Literatur, die Bezeichnung OR, obwohl XOR präziser wäre, da nie mehr als ein direkter Unterzustand aktiv sein kann.

Zusammengesetzte Zustände (Typ AND)

Dieser Zustand ist wie der Typ OR-Zustand aufgebaut, besitzt darüberhinaus jedoch noch eine Partitionierung der Unterzustände. Wenn dieser Zustand aktiv ist, muss in jeder Partition genau ein Zustand aktiv sein.

1.1.2 Übergänge (Transitionen)

Durch Übergänge wird festgelegt, welche Zustände nach dem Eintreten gewisser Bedingungen aktiviert bzw. deaktiviert werden. Übergänge bestehen aus:

- einer nichtleeren Menge von Quellzuständen
- einer nichtleeren Menge von Zielzuständen
- einem Ereignis
- einer Bedingung
- den beim Übergang erzeugten Aktionen

Wenn die Bedingung erfüllt ist, alle Quellzustände aktiv sind und das Ereignis im vorherigen Schritt generiert wurde, werden die Quellzustände deaktiviert, die Zielzustände aktiviert und die entsprechenden Aktionen der Transition ausgeführt.

Bedingungen

Eine Bedingung prüft verschiedene Gegebenheiten (Randbedingungen), die sich auf die Zustände oder auf externe Einflüsse beziehen. Ausschlaggebend für eine Bedingung ist, dass sie einen Wahrheitswert hat. Beschrieben wird sie durch eine aussagenlogische Formel.

Ereignisse

Ereignisse sind Randbedingungen, die aktiv generiert werden und nach einem Zeitschritt selbständig wieder verschwinden. Sie können nur durch Aktionen erzeugt und durch Übergänge getestet werden. Ereignisse sind notwendig, um einen Übergang zu initiieren.

Aktionen

Aktionen werden bei einem durchgeführten Übergang ausgeführt. Man unterscheidet zwischen

- internen Aktionen (Generieren eines Ereignisses, Löschen der History eines Zustands, etc.), die sich auf die Elemente des Statecharts auswirken.
- externen Aktionen (Interaktion mit der Benutzerschnittstelle, etc.)

1.2 Formale Definition

Vorbemerkung Bevor wir mit der formalen Definition beginnen, wollen wir noch kurz erklären, dass für uns $\mathbb{N} = \{1, 2, 3, \dots\}$ gilt.

1.2.1 Statechart

Formal können wir ein *Statechart* als ein Tupel $SC = (Z, \sigma, \pi, d, H, K, E, A, L_B)$ betrachten. Hierbei ist

- Z die Menge der Zustände,
- σ die Unterzustandsfunktion,
- π die Partitionsfunktion,
- d die Defaulteinstiegsfunktion,
- H die Menge der Historyeinstiege mit deren Historyfunktionen,
- K die Menge der Kanten,
- E die Menge der Ereignisse,
- A die Menge der Aktionen und
- L_B eine aussagenlogische Sprache zur Beschreibung der Bedingungen.

Bemerkung. Im Gegensatz zu einem endlichen Automaten fehlt hier das Analogon zur Zustandsübergangsfunktion. Diese ist für Statecharts nicht einfach festlegbar. Sie ist in den Kanten impliziert. Wir werden das Verhalten über der Zeit im Abschnitt 1.4 beschreiben.

1.2.2 Die Menge der Zustände Z

Die *Menge der Zustände* $Z = \{z_1, \ldots, z_{n_Z}\}$ mit $n_Z \in \mathbb{N}$ ist eine endliche Menge von unterscheidbaren Elementen.

1.2.3 Die Unterzustandsfunktion σ

Für jedes $z \in Z$ ist $\sigma(z) \subset Z$ die *Menge der direkten Unterzustände* von z.

Definition (Atomare Zustände).

(1.1) z ist *atomarer Zustand* gdw. $\sigma(z) = \emptyset$.

Definition (Eindeutigkeit des Oberzustands).
Es muß gelten

(1.2) $x \in \sigma(z) \Rightarrow (\forall y \in Z : x \in \sigma(y) \Leftrightarrow y = z)$.

Lemma (Fixpunktfreiheit).
Die Unterzustandsfunktion ist fixpunktfrei, das heißt:

$$\forall z \in Z : z \notin \sigma(z). \tag{1.3}$$

Beweis siehe 1.9.

Definition (Transitive Hülle der Unterzustände).
Wir definieren induktiv die transitive Hülle der Unterzustandsfunktion: Für $z \in Z$ ist

$$\sigma^0(z) = \{z\} \tag{1.4}$$

$$\sigma^{i+1}(z) = \bigcup_{y \in \sigma^i(z)} \sigma(y) \tag{1.5}$$

$$\sigma^+(z) = \bigcup_{i=1}^{\infty} \sigma^i(z) \tag{1.6}$$

$$\sigma^*(z) = \bigcup_{i=0}^{\infty} \sigma^i(z) \tag{1.7}$$

Definition (Fixpunktfreiheit der transitiven Hülle).
Es gilt:

$$\forall z \in Z : z \notin \sigma^+(z). \tag{1.8}$$

Beweis der Fixpunktfreiheit von σ.

$$\begin{aligned} (1.8) \Rightarrow (1.3): \quad & \sigma^+(z) \\ \overset{\text{n. Def. (1.6)}}{=} \quad & \bigcup_{i=1}^{\infty} \sigma^i(z) \\ = \quad & \sigma(z) \cup \bigcup_{i=2}^{\infty} \sigma^i(z) \end{aligned} \tag{1.9}$$

Es ist also $\sigma(z) \subseteq \sigma^+ z$. Damit folgt aus $z \notin \sigma^+(z) \Rightarrow z \notin \sigma(z)$. □

Korollar (Direktheit).
Es gilt

$$\forall z \in Z : x \in \sigma^2(z) \Rightarrow x \notin \sigma(z) \tag{1.10}$$

oder in einer anderen Formulierung:

$$\forall z \in Z : x \in \sigma(z) \Rightarrow x \notin \sigma^i(z) \text{ f. a. } i > 1. \tag{1.11}$$

Das heißt, dass ein Zustand x, der in einer Tiefe von zwei unter z liegt, nicht auch direkter Nachfolger von z sein kann.

Beweis der Direktheit.

Annahme: $\exists x, z \in Z : x \in \sigma^2(z) \wedge x \in \sigma(z)$.
Mit $x \in \sigma^2(z)$ gilt also $\exists y \in \sigma(z) : x \in \sigma(y)$. Nach Annahme ist auch $x \in \sigma(z)$. Aus $x \in \sigma(y)$ und $x \in \sigma(z)$ folgt dann mit (1.2), dass $y = z$. Da $y \in \sigma(z)$ war, heißt das $z \in \sigma(z)$. Dies steht aber im Widerspruch zu (1.3). □

Der Beweis der anderen Formulierung verläuft ähnlich.

Korollar (Symmetriefreiheit von σ).
Aus (1.8) folgt direkt:

$$\forall y, z \in Z : y \in \sigma(z) \Rightarrow z \notin \sigma(y). \tag{1.12}$$

$$\text{bzw. } \forall z \in Z : z \notin \sigma^2(z). \tag{1.13}$$

Definition (Tiefe).
Die *Tiefe eines Zustands z* ist definiert durch

$$\text{depth}(z) = \begin{cases} 0, & \text{wenn } \nexists y \in Z : z \in \sigma(y) \\ \max\{i \in \mathbb{N} \mid \exists y \in Z : z \in \sigma^i(y)\}, & \text{sonst} \end{cases} \tag{1.14}$$

Ein Zustand z mit $\text{depth}(z) = 0$ heißt *Zustand auf oberster Ebene* oder *Wurzelzustand.* Der Wurzelzustand ist eindeutig, d. h. $\forall x, y \in Z : \text{depth}(x) = \text{depth}(y) = 0 \Rightarrow x = y$.

1.2.4 Die Partitionsfunktion π

Für jedes $z \in Z$ ist $\pi(z)$ eine vollständige Zerlegung von $\sigma(z)$ in disjunkte Teilmengen *(Partitionen)* $\pi_1(z), \ldots, \pi_{n_{\pi(z)}}(z)$ mit $n_{\pi(z)} \in \mathbb{N}$. Das heißt, für je zwei Elemente $x \in \pi_i$ und $y \in \pi_j$ mit $i \neq j$ gilt $x \neq y$.

$n_{\pi(z)}$ heißt *Anzahl der Partitionen* von $\sigma(z)$.

Definition (Partitionen der atomaren Zustände).
Für alle atomaren Zustände $z \in Z$ mit z ist $\sigma(z) = \emptyset$. Es ist deshalb $\pi(z) = (\emptyset)$ die disjunkte Zerlegung in eine Partition. Wir schreiben auch kurz $\pi(z) = \emptyset$.

Definition (Partition der nichtatomaren Zustände).
Für alle nichtatomaren Zustände $z \in Z$ soll gelten:

$$\forall \pi_i(z) \in \pi(z) : \pi_i(z) \neq \emptyset. \tag{1.15}$$

Bemerkung (Partitionierung aller Unterzustände).
Wegen der Vollständigkeit der Zerlegung liegen alle Unterzustände in einer Partition, das heißt:

$$\forall z \in Z : \left(y \in \sigma(z) \Leftrightarrow \exists i \in \{1, \ldots n_{\pi(z)}\} : y \in \pi_i(z)\right). \tag{1.16}$$

Definition (Charakterisierung zusammengesetzter Zustände vom Typ OR).
Ein Zustand $z \in Z$ heißt *zusammengesetzter Zustand vom Typ* OR

$$\text{gdw. } \sigma(z) \neq \emptyset \wedge n_{\pi(z)} = 1.$$

Definition (Charakterisierung zusammengesetzter Zustände vom Typ AND).
Ein Zustand $z \in Z$ heißt *zusammengesetzter Zustand vom Typ* AND

$$\text{gdw. } \sigma(z) \neq \emptyset \wedge n_{\pi(z)} > 1.$$

Definition (Erweiterung der Partitionsfunktion).
Wir definieren die Erweiterung der Partitionsfunktion wie folgt. Für $z \in Z$ ist

$$\pi_i^*(z) = \bigcup_{y \in \pi_i(z)} \sigma^*(y) \tag{1.17}$$

$$\pi^*(z) = (\pi_1^*(z), \ldots, \pi_{n_{\pi(z)}}^*(z)) \tag{1.18}$$

Das heißt, in $\pi_i^*(z)$ liegen alle Elemente der Partition $\pi_i(z)$, sowie deren sämtliche (auch indirekten) Unterzustände. Damit ist π^* eine Partitionierung sämtlicher Unterzustände eines Zustands.

Beispiel.

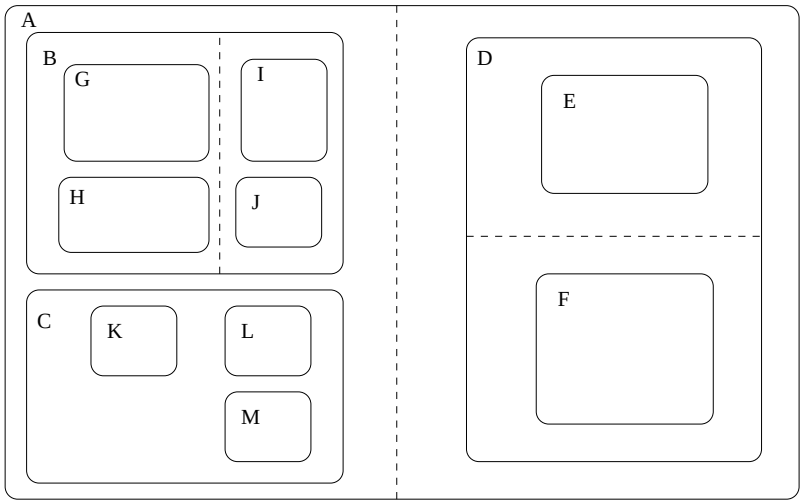


Abbildung 1.1: Beispiel-Statechart zur Erläuterung von σ und π

Um die Funktionen π und σ zu erläutern,[1] betrachten wir Abbildung 1.1. Die Menge der Zustände ist hier

$$Z = \{A, B, C, D, E, F, G, H, I, J, K, L, M\}.$$

Für alle $z \in Z$ sind nun π und σ wie folgt definiert:

[1] Die Erläuterung der grafischen Definition folgt in Abschnitt 1.3.

z	Typ	$\sigma(z)$	$\sigma^+(z)$	$\pi(z)$	$\pi^*(z)$
A	AND	$\{B,C,D\}$	$Z\backslash\{A\}$	$(\{B,C\},\{D\})$	$(\{B,G,H,I,J,C,K,L,M\}, \{D,E,F\})$
B	AND	$\{G,H,I,J\}$		$(\{G,H\},\{I,J\})$	
C	OR	$\{K,L,M\}$		$(\{K,L,M\})$	
D	AND	$\{E,F\}$		$(\{E\},\{F\})$	
E	ATOM	$\emptyset$			
F	ATOM				
G	ATOM				
H	ATOM				
I	ATOM				
J	ATOM				
K	ATOM				
L	ATOM				
M	ATOM				

1.2.5 Die Defaulteinstiegsfunktion d

Mit Hilfe der Defaulteinstiege wird festgelegt, welche Unterzustände von zusammengesetzten Zuständen aktiviert werden sollen, wenn der komplexe Zustand aktiviert wird und dabei für einige Partitionen kein aktivierter Zustand angegeben ist.

Die Defaulteinstiegsfunktion d ordnet jedem Zustand $z \in Z$ die Menge seiner Defaulteinstiege zu. Für atomare Zustände ist $d(z) = \emptyset$. Für komplexe Zustände ist $d(z) \subseteq \sigma(z)$, wobei aus jeder Partition von $\pi(z)$ genau ein Element in $d(z)$ enthalten sein muß. Die Mächtigkeit der Defaulteinstiegsmenge $d(z)$ ist also $n_{\pi(z)}$.

1.2.6 Die Historyfunktion h und die Menge der Historyeinstiege H

Jedem komplexen Zustand z kann ein Historyeinstieg $h \in H$ zugeordnet werden. Wir bezeichnen den Historyeinstieg des Zustands z_i mit h_i. Analog zu d ordnet die Historyfunktion h diesem Zustand $z \in Z$ zu jedem Zeitpunkt $t \in \mathbb{N}$ eine Menge von Einstiegszuständen $y \in \sigma(z)$, welche die letzten aktivierten Zustände sind, zu. Es ist also eine Funktion definiert auf $Z \times \mathbb{N} \to \wp(Z)$. Der Funktionswert kann jedoch nicht im voraus spezifiziert werden, da sich die Werte der Funktion während der Laufzeit ergeben. Siehe hierzu Abschnitt 1.4 ab Seite 31.

Für die Zeit $t = 1$ wird vereinbart:

$$\forall h_i \in H : h(z_i, 1) = d(z_i). \tag{1.19}$$

1.2.7 Die Menge der Ereignisse E

Die Menge der möglichen Ereignisse ist $E = \{e_1, \ldots, e_{n_E}\}$ mit $n_E \in \mathbb{N}, n_E \geq 2n_Z$. Je zwei $e_i \in E$ sind verschieden.

Um die unterschiedlichen Ereignisse zu trennen, partitionieren wir die Menge der Indizes $\{1,\dots n_E\}$ wie folgt:

- $EA = \{1,\dots,n_Z\}$ für die Ereignisse "Zustand aktiviert".
- $ED = \{n_Z+1,\dots,2n_Z\}$ für die Ereignisse "Zustand deaktiviert".
- $EI = \{2n_Z+1,\dots,2n_Z+n_{EI}\}$ für weitere interne Ereignisse
- $EX = \{2n_Z+n_{EI}+1,\dots,2n_Z+n_{EI}+n_{EX}\}$ für externe Ereignisse.

Die Semantik der Ereignisse legen wir nun wie folgt fest:

- Für alle $i \in EA$ ist e_i das Ereignis, dass z_i aktiviert wurde.
- Für alle $i \in ED$ ist e_i das Ereignis, dass z_{i-n_Z} deaktiviert wurde.
- Für alle $i \in EI$ ist e_i ein intern generiertes Ereignis (z. B. durch eine Transition generiert).
- Für alle $i \in EX$ ist e_i ein extern generiertes Ereignis (z. B. durch ein Dialogelement generiert).

1.2.8 Die Menge der Aktionen A

Die Menge der Aktionen $A = \{a_1,\dots,a_{n_A}\}$ für $n_A \in \mathbb{N}, n_A \geq n_E$ bezeichnet die im Statechart möglichen Aktionen. Für alle $i \leq n_E$ ist a_i die Aktion, das Ereignis e_i zu erzeugen. Alle a_i mit $i > n_E$ beschreiben Aktionen, die keine Ereignisse erzeugen, zum Beispiel das Löschen einer History oder Kommunikation mit der Benutzerschnittstelle.

1.2.9 Die Menge der Kanten K

Eine Kante $k \in K$ im Statechart wird durch ein Tupel (V,N,e,φ,Δ) beschrieben. Hierbei ist $\emptyset \neq V \subseteq Z$ die Menge der Startzustände und $\emptyset \neq N \subseteq (Z \cup H)$ die Menge der Zielzustände. e ist das Ereignis, das den Übergang auslöst. φ ist eine aussagenlogische Formel mit $\varphi \in L_B$ und $\Delta \subseteq A$ ist die Teilmenge der dabei erzeugten Aktionen.

Definition (Übergangsbedingung).
Ein *Übergang*[2] findet in einem Zeitschritt statt gdw. e ist eingetreten, alle $v \in V$ sind aktiv und φ ist erfüllt.

Integritätsbedingungen für einen Übergang

Kantenanfang von einem Unterzustand Wenn $\#V > 1$, dann ex. $z \in Z$, so dass $\forall v \in V : v \in \sigma^+(z)$ und es gilt:

(1.20) $$\forall v,w \in V : v \in \pi_i^*(z) \wedge w \in \pi_j^*(z) \wedge v \neq w \Rightarrow i \neq j.$$

wobei $\pi^*(z) = (\pi_1^*(z),\dots,\pi_{n_{\pi(z)}}^*(z))$.

Das heißt: Wenn es mehr als einen Startzustand gibt, dann existiert ein gemeinsamer (möglicherweise nicht direkter) Oberzustand z, so dass alle Startzustände in verschiedenen Partitionen von z liegen.

[2] vgl. Abschnitt 1.4.2 auf Seite 32 für eine genauere Definition

Kantenende auf einem Unterzustand Wenn $\#N > 1$, dann ex. $z \in Z$, so dass $\forall v \in N : v \in \sigma^+(z)$ und es gilt:

$$\forall v, w \in N : v \in \pi_i^*(z) \wedge w \in \pi_j^*(z) \wedge v \neq w \Rightarrow i \neq j. \tag{1.21}$$

wobei $\pi^*(z) = (\pi_1^*(z), \ldots, \pi_{n_{\pi(z)}}^*(z))$.

Das heißt: Wenn es mehr als einen Zielzustand gibt, dann existiert ein gemeinsamer (möglicherweise nicht direkter) Oberzustand z, so dass alle Zielzustände in verschiedenen Partitionen von z liegen.

Bemerkung. Wenn v oder w aus H sind, gilt analog dass sie in verschiedenen Partitionen liegen müssen, auch wenn dies nicht explizit durch diese Formel erfasst wurde.

Bemerkungen.

1. $\#V > 1$ impliziert, dass die Anfangspunkte des Übergangs in einem komplexen Zustand vom Typ AND liegen.
2. $\#N > 1$ impliziert, dass die Endpunkte des Übergangs in einem komplexen Zustand vom Typ AND liegen.
3. Die folgenden Sonderfälle können auftreten:
   - $V \cap N \neq \emptyset$. Das heißt, dass es einen Pfeil gibt, bei dem ein Zielzustand ein Startzustand war.
   - $\Delta = \emptyset$. Das heißt, dass keine Aktion bei einem Übergang ausgeführt wird.
   - $\models \varphi$. Das heißt, dass der Übergang bedingungslos ausgeführt wird.
   - $e =$ „Zustand z_i aktiviert“ für $z_i \in V$. Das heißt, dass das Ereignis, den Zustand zu verlassen von diesem Zustand selbst erzeugt wurde. Diese Konstruktion ist also ein Analogon zu einem transienten Zustand, also einem Zustand, der garantiert im nächsten Zeitschritt verlassen wird, wenn die immer wahre Kantenbedingung angegeben ist.[3]

1.2.10 Die Sprache der Bedingungen L_B

Die *Sprache der Bedingungen* L_B ist eine Sprache der Aussagenlogik $\mathcal{L}_A$. Diese besteht – wie jede aussagenlogische Sprache – aus:

1. aussagenlogischen Variablen p_m f. a. $m \in \mathbb{N}$
2. den logischen Zeichen $\neg$ und $\vee$
3. den technischen Zeichen (und).

[3] ansonsten würde dieser Zustand nie wieder verlassen werden

Bemerkungen.

- Die aussagenlogischen Variablen p_i repräsentieren den Wahrheitswert von Bedingungen, die im Statechart getestet werden sollen, wie zum Beispiel die Frage, ob ein Zustand aktiv ist.

- Weitere logische Operationen (wie $\Rightarrow$, $\wedge$ etc.) folgern wir aus $\neg$ und $\vee$ in der üblichen Weise.

1.3 Grafische Definition

In diesem Abschnitt wollen wir die grafische Darstellung der eben eingeführten Elemente definieren.

1.3.1 Zustände

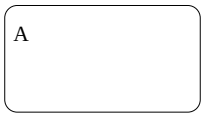


Abbildung 1.2: Atomarer Zustand mit Identifikation A

Zustände werden durch an den Ecken abgerundete Rechtecke dargestellt. Bei atomaren Zuständen enthalten diese nur die Identifikation, ansonsten auch die Unterzustände (siehe Abbildungen 1.2 und 1.3). Bei Zuständen vom Typ ANDkann die Identifikation auch direkt über dem Zustand stehen.

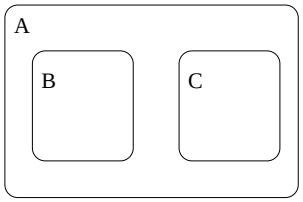


Abbildung 1.3: Zusammengesetzter Zustand (Typ OR)

Die Partitionierung der zusammengesetzten Zustände vom Typ AND werden durch gestrichelte Linien an den Partitionsgrenzen dargestellt (siehe Abbildung 1.4). Keine zwei Zustände dürfen sich überlappen, und kein Zustand darf eine Partitionsgrenze schneiden. Dies entspricht der formalen Definition der Eindeutigkeit des Oberzustands (siehe Gleichung (1.2)) sowie der disjunkten Zerlegung von Unterzuständen durch π (siehe Abschnitt 1.2.4).[4]

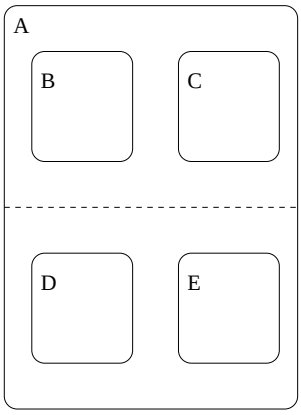


Abbildung 1.4: Zusammengesetzter Zustand mit Partitionierung (Typ AND)

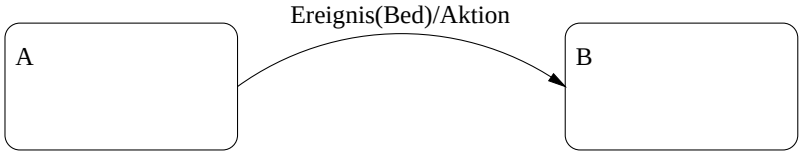


Abbildung 1.5: Ein einfacher Übergang von A nach B

1.3.2 Übergänge

Die Übergänge werden durch Pfeile zwischen den Zuständen dargestellt. Die Pfeile sind beschriftet mit `Ereignis(Bedingung)/Aktion` (siehe Abbildungen 1.5 und 1.7). Wenn die Bedingung die immer wahre Bedingung ist, kann sie mit den Klammern entfallen. Wenn die auszuführende Aktionsmenge leer ist, kann sie mit dem Schrägstrich auch entfallen.

Die Pfeile dürfen keine Partitionsgrenzen schneiden. Sie dürfen Zustandsränder nur dann schneiden, wenn sie einen inneren Zustand eines zusammengesetzten Zustands erreichen wollen oder von einem solchen ausgehen.

Zur Erläuterung betrachten wird die Abbildung 1.6. Die Transition unter dem Ereignis $E1$ ist fehlerhaft, denn sie schneidet eine Partitionsgrenze. Semantisch würde also der Zustand A deaktiviert und B aktiviert werden. Dies bedeutet jedoch, dass in der linken Partition überhaupt kein Zustand mehr aktiv ist! Dieser Fall darf nach Definition der Statecharts nicht auftreten.

Hingegen ist die Transition unter dem Ereignis $E2$ korrekt. Wird der Oberzustand verlassen und anschließend wieder betreten, so wird der Zustand B aktiviert. Da der Oberzustand betreten wurde, werden in allen Partitionen, in denen kein Zustand aktiviert wird, die Defaulteinstiege benutzt. Somit ist auch A aktiviert und keine Partition ohne aktiven Zustand.

[4] Eine Diskussion über eine andere Definition findet sich in Abschnitt 2.2.4

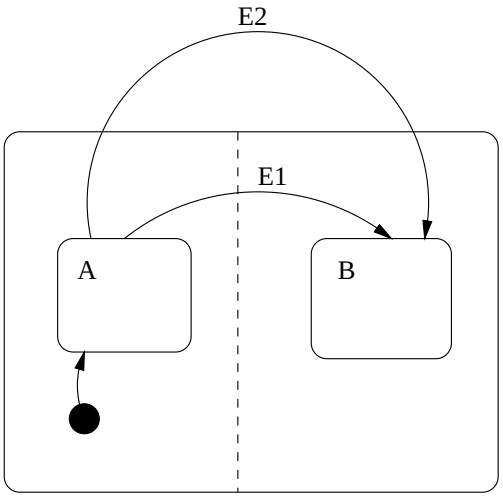


Abbildung 1.6: Fehlerhaftes Statechart, da Partitionsgrenzen geschnitten werden.

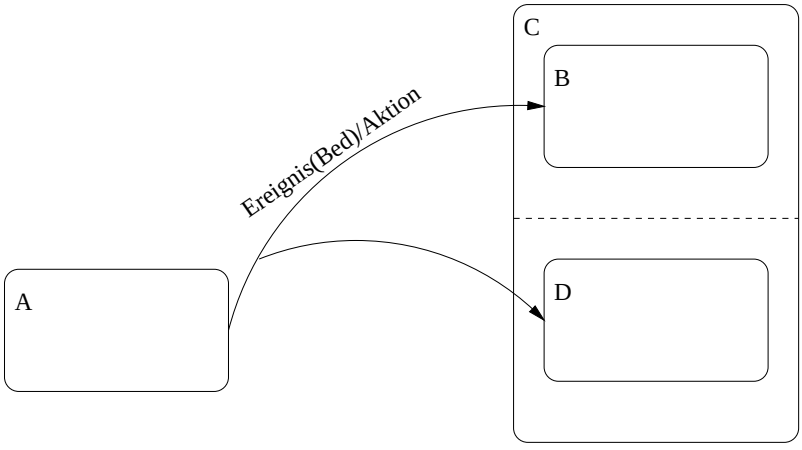


Abbildung 1.7: Ein komplexer Übergang

1.3.3 Defaulteinstieg

Defaulteinstiege werden durch einen gefüllten Kreis dargestellt, der genau eine ausgehende und keine eingehende Kante hat. Das Ziel dieser Kante muss in derselben Partition liegen wie der Defaulteinstieg selbst. Siehe hierzu die Abbildungen 1.8 und 1.9.

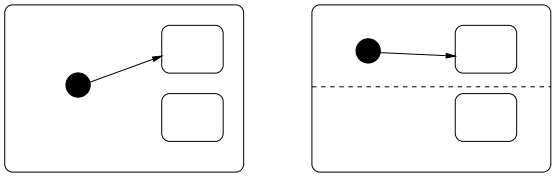


Abbildung 1.8: Korrekte Defaulteinstiege

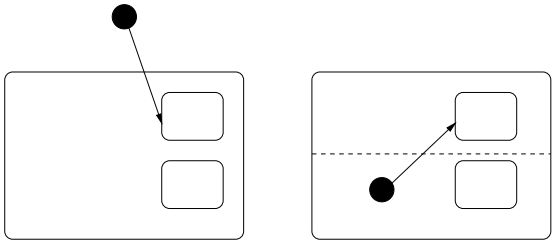


Abbildung 1.9: Fehlerhafte Defaulteinstiege

1.3.4 Historyeinstieg

Der Historyeinstieg wird durch ein nichtgefüllten Kreis mit einem H dargestellt. Der Historyeinstieg darf eingehende Kanten haben (um den Historyeinstieg zu benutzen) und ausgehende, welche benutzt werden, wenn keine History vorhanden ist.[5]

Ziele von Kanten, die von Historyeinstiegen ausgehen, müssen, wie bei den Defaulteinstiegen, in denselben Partitionen liegen wie der Historyeinstieg selbst. Siehe Abbildungen 1.10 und 1.11.

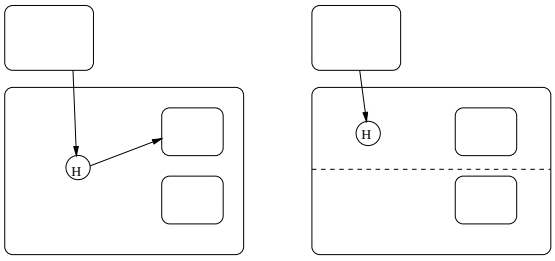


Abbildung 1.10: Korrekte Historyeinstiege

[5] In dieser Definition ist dies der Einfachheit halber nicht beachtet. Andere Modelle, wie auch unsere Implementation, erlauben dies jedoch. Siehe hierzu auch Abschnitt 2.3.2.

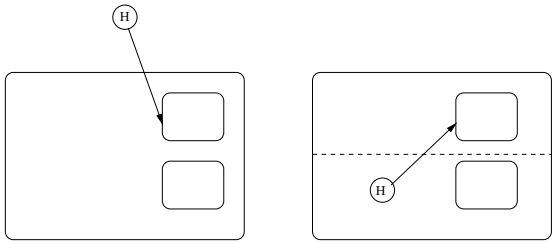


Abbildung 1.11: Fehlerhafte Historyeinstiege

1.3.5 Konnektoren

Konnektoren werden grafisch durch nichtausgefüllte Kreise dargestellt. Für sie gilt bezüglich der Platzierung und der Anbindung an Kanten das für Zustände Gesagte. Wir verzichten hier auf eine genauere Definition, da sie auch in unserem theoretischen Modell nicht definiert sind.

1.4 Temporale Auswertung

In diesem Abschnitt soll erläutert werden, wie sich ein Statechart während seiner Abarbeitung verhält und wie sich seine Konfiguration ändert.

1.4.1 Der Zustand eines Statecharts

Die Konfiguration eines Statecharts ist abhängig von der Zeit. Wir benutzen auch hier eine diskrete Zeit $t \in \mathbb{N}$ ein. Die komplette Konfiguration eines Statecharts lässt sich als ein Vektor beschreiben, dessen Elemente wir unten konstruieren werden. Damit kann man das Verhalten des Statecharts über die Zeit als eine Folge von Vektoren darstellen. Es gilt nun, diese Vektoren zu bestimmen. Folgende Werte ändern sich über die Zeit:

- welche Zustände aktiviert bzw. deaktiviert sind
- welche Ereignisse gerade ausgelöst wurden
- welchen Wert die Historyeinstiege haben
- welche Bedingungen welchen Wahrheitswert haben

Den ersten und den letzten Punkt haben wir bereits in der formalen Definition der Sprache L_B in den aussagenlogischen Variablen p_m zusammengefasst (siehe Abschnitt 1.2.10). Dies sind zwar nach Definition unendlich viele, jedoch reicht immer eine endliche Teilmenge zur Beschreibung des Systems, indem man nur die p_i betrachtet, die in den benutzten Formeln des Statecharts vorkommen. Wir nummerieren diese p_i um und erhalten den Vektor $(\tilde{p}_1, \ldots, \tilde{p}_{n_{\tilde{p}}})$. Analog erhalten wir aus den Ereignissen einen Ereignisvektor $(\tilde{e}_1, \ldots, \tilde{e}_{n_{\tilde{e}}})$ und ergänzen diesen noch um die Werte der Historyeinstiege.

Die Konfiguration eines Statecharts ist also

$$\Gamma(t) = (\tilde{p}_1(t), \ldots, \tilde{p}_{n_{\tilde{p}}}(t), \tilde{e}_1(t), \ldots, \tilde{e}_{n_{\tilde{e}}}(t), h(z_1,t), \ldots, h(z_{n_Z},t)). \tag{1.22}$$

Im folgenden schreiben wir auch

$$z_i(t) = \begin{cases} 1, & \text{wenn der Zustand } z_i \text{ aktiv ist,} \\ 0, & \text{wenn der Zustand } z_i \text{ inaktiv ist.} \end{cases}$$

Bemerkung. Man beachte, dass einige der $\tilde{p}_i(t)$ mit einem $z_j(t)$ identisch sind, wenn in Bedingungen getestet wird, ob ein Zustand j aktiviert oder nicht aktiviert ist (vgl. Abschnitt 1.2.10).

1.4.2 Der Zeitschritt

Wird nun ein Zeitschritt durchgeführt, also $t \curvearrowright t+1$, so werden mittels der Übergänge K aus $\Gamma(t)$ die Folgezustände ermittelt. Dadurch und durch die nicht vom Statechart beeinflussbaren Randbedingungen sind bereits $\tilde{p}_1(t+1), \ldots, \tilde{p}_{n_{\tilde{p}}}(t+1)$ festgelegt. Die Werte für die Ereignisse werden neu aus den jetzt vollzogenen Übergängen sowie extern generierten Ereignissen gesetzt, und aus den alten Zuständen werden die Historyinformationen ermittelt. Danach ist der Vektor $\Gamma(t+1)$ vollständig definiert.

Im Folgenden betrachten wir eine Kante $k \in K$ als Tupel $(V, N, e, \varphi, \Delta)$. Siehe hierzu auch Abschnitt 1.2.9 auf Seite 25. Wir legen weiterhin fest, dass wir mit $z_i \in N$ explizit einen Zustand meinen. Für Historyeinstiege schreiben wir $h_i \in N$.

Ermittlung der Folgezustände

Sei $\Gamma(t)$ nun der Statechart-Konfigurationsvektor zum Zeitpunkt t. Im Folgezustandsvektor $Z(t+1) = (z_1(t+1), \ldots, z_{n_Z}(t+1))$ gilt nun

$$\begin{aligned}
(1.23) \quad & \exists k \in K : (z_i \in N \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \quad \Rightarrow \quad z_i(t+1) = 1 \\
(1.24) \quad & \nexists k \in K : (z_i \in V \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \\
& \wedge \ \nexists k \in K : (z_i \in N \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \quad \Rightarrow \quad z_i(t+1) = z_i(t) \\
(1.25) \quad & \exists k \in K : (z_i \in V \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \\
& \wedge \ \nexists k \in K : (z_i \in N \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \quad \Rightarrow \quad z_i(t+1) = 0.
\end{aligned}$$

Wenn das Ziel kein Zustand, sondern ein Historyeinstieg $h \in H$ ist, gilt analog folgende Bedingung:

$$\begin{aligned}
(1.26) \quad & \exists k \in K : (h_i \in N \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \\
\Rightarrow \quad & \forall y \in h(z_i, t) : y(t+1) = 1
\end{aligned}$$

Diese Gleichungen gelten exklusive der Setzung durch Unter- oder Oberzustände.

Durch diese Gleichungen wird festgelegt, dass

- wenn es eine Kante gibt, die nach z_i führt, deren sämtlichen Startzustände aktiviert sind, das auslösende Ereignis eingetreten und die Bedingung wahr ist, dann wird im nächsten Zeitschritt z_i aktiviert. (1.23)
- wenn es keine Kante mit obigen Bedingungen gibt und keine Kante von z_i abgeht, deren sämtliche Startzustände aktiviert sind, das Ereignis eingetreten und die Bedingung wahr ist, dann bleibt die Aktivierung bzw. Deaktivierung des Zustands z_i im nächsten Zeitschritt unverändert. (1.24)
- wenn es keine Kante wie aus (1.23) gibt, jedoch eine Kante, von der z_i Startzustand ist, alle Startzustände aktiviert, das Ereignis eingetreten und die Bedingung wahr ist, dann wird z_i im nächsten Zeitschritt deaktiviert. (1.25)
- wenn der Zielzustand ein Historyeinstieg ist, dann werden die Zustände, auf die der Historyeinstieg zeigt, im nächsten Zeitschritt aktiviert. (1.26)

Ermittlung der Folgeereignisse

Für die Ereignisse $e_i \in E$ mit $i \in EA \cup ED$ gelten folgende temporale Bedingungen:

$$\text{(1.27)} \quad \forall i \in \{1,\dots,n_Z\} : z_i(t) = 0 \wedge z_i(t+1) = 1 \quad \Rightarrow \quad e_i(t+1) = 1$$
$$\text{(1.28)} \quad \forall i \in \{1,\dots,n_Z\} : z_i(t) = 1 \wedge z_i(t+1) = 0 \quad \Rightarrow \quad e_{i+n_Z}(t+1) = 1$$
$$\text{(1.29)} \quad \exists k \in K : (z_i \in N \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \quad \Leftrightarrow \quad e_i(t+1) = 1$$
$$\text{(1.30)} \quad \exists k \in K : (z_i \in V \wedge (\forall v \in V : v(t) = 1) \wedge \varphi \wedge e) \quad \Leftrightarrow \quad e_{i+n_Z}(t+1) = 1.$$

Auch diese Gleichungen gelten exklusive der Setzung durch Ober- oder Unterzustände.

Durch diese Gleichungen wird festgelegt, dass

- wenn zu einem Zeitpunkt t ein Zustand inaktiv ist und im Zeitpunkt $t+1$ der selbe Zustand aktiv ist, dann das Ereignis e_i (Zustand i aktiviert) erzeugt wird. (1.27)
- wenn zu einem Zeitpunkt t ein Zustand aktiv ist und im Zeitpunkt $t+1$ der selbe Zustand inaktiv ist, dann das Ereignis e_{i+n_Z} (Zustand i deaktiviert) erzeugt wird. (1.28)
- das Ereignis e_i (Zustand i aktiviert) genau dann erzeugt wird, wenn es eine Kante gibt, von der z_i Zielzustand ist, alle Startzustände vorher aktiv waren, das Ereignis e erzeugt wurde und die Bedingung wahr ist. (1.29)
- das Ereignis e_{i+n_Z} (Zustand i deaktiviert) genau dann erzeugt wird, wenn es eine Kante gibt, von der z_i Startzustand ist, alle Startzustände vorher aktiv waren, das Ereignis e erzeugt wurde und die Bedingung wahr ist. (1.30)

Bemerkungen.

1. Insbesondere folgt hieraus, dass wenn $z_i \in V$ und $z_i \in N$, also eine Schleife auf einem Zustand benutzt wird, sowohl das Ereignis e_i (Zustand i aktiviert) als auch das Ereignis e_{i+n_Z} (Zustand i deaktiviert) erzeugt werden.

2. Die Formeln (1.27) und (1.28) stellen nicht nur Folgerungen der Formeln (1.29) und (1.30) dar, da ein Aktivieren bzw. Deaktivieren von Zuständen in einem Zeitschritt nicht nur durch einen direkten Kantenübergang ausgelöst werden kann, sondern auch durch indirekte Ober- oder Unterzustandsaktivierungen oder durch Historyeinstiege.

Ermittlung aktivierter Unterzustände

Wenn ein komplexer Zustand aktiviert wird, wird automatisch auch ein Unterzustand aktiviert. Bei komplexen Zuständen vom Typ AND wird sogar in jeder Partition ein Unterzustand aktiviert. Formal können wir dieses Verhalten wie folgt erfassen

$$\forall i \in \{1,\ldots,n_Z\} : e_i(t+1) = 1 \Rightarrow \forall y \in d(z_i) : y(t+1) = 1 \tag{1.31}$$

Die Historyeinträge werden auf den aktuellen Stand gesetzt, das heißt:

$$\forall z \in Z : \forall y : z \in \sigma(y) \wedge z(t) = 1 \Leftrightarrow z \in h(y,t+1) \tag{1.32}$$

Diese Formeln gelten natürlich rekursiv. Zur Erinnerung: Die Historyfunktion ist am Anfang mit dem Defaulteinstiegswert initialisiert. Hier gibt es in der Literatur auch andere Varianten.[6]

Ermittlung deaktivierter Unterzustände

Wenn ein Zustand deaktiviert wird, werden auch alle Unterzustände deaktiviert, das heißt:

$$\forall z \in Z : z(t) = 0 \Rightarrow \forall y \in \sigma^+(z) : y(t) = 0 \tag{1.33}$$

Die Historyeinträge bleiben dabei unverändert, das heißt:

$$\forall i \in \{1,\ldots,n_Z\} : e_{i+n_Z}(t+1) = 1 \Rightarrow h(z_i,t+1) = h(z_i,t) \tag{1.34}$$

Ermittlung aktivierter Oberzustände

Wenn ein Unterzustand direkt angesprungen wird, dann werden automatisch alle Oberzustände aktiviert.

$$\exists z \in Z : z(t) = 1 \Rightarrow \forall y \in Z : z \in \sigma^+(y) : y(t) = 1 \tag{1.35}$$

[6] siehe hierzu Abschnitt 2.3.2

Ermittlung deaktivierter Oberzustände

Wenn alle Unterzustände eines Oberzustands deaktiviert wurden, dann wird auch der Oberzustand deaktiviert.

$$\forall z \in Z : (\forall y \in \sigma(z) : y(t) = 0) \Leftrightarrow z(t) = 0 \tag{1.36}$$

Bemerkung. Diese Formel stellt eine Erweiterung der Gleichung in (1.33) dar.

1.5 Abweichungen

Unser Modell beschreibt formal nicht, dass auch Kanten von Historyeinstiegen ausgehen können. Dieses Konzept, welches also auch $h \in V$ für eine Kante k vorsieht, erweitert das formale Modell nicht wesentlich, da dies nur ein weiterer Defaulteinstieg ist, der benutzt wird, wenn noch keine History vorhanden ist. Um diese formale Definition übersichtlicher zu halten, haben wir uns entschlossen, die Historyfunktion mit dem Defaulteinstieg zu initialisieren. Wir umgehen so viele Fallunterscheidungen sowie Abfragen, ob überhaupt eine History vorhanden ist, setzen jedoch voraus, dass jeder komplexe Zustand einen Defaulteinstieg hat.

πάντα ῥεῖ[1]
(Heraklit)

Kapitel 2

Varianten von Statecharts in der Literatur

2.1 Einleitung

In diesem Kapitel sollen die Unterschiede einzelner Fassungen von Statecharts erklärt werden, die in der Literatur vorgestellt werden. Unser besonderes Augenmerk gilt dabei [HN96], denn einer der beiden Autoren ist gerade jener David Harel, der 1987 den Formalismus der Statecharts zum ersten Mal vorgeschlagen hat.[2] Desweiteren werden wir die Art und Weise betrachten, wie Statecharts im Programm Statemate implementiert wurden. Laut Harel, der an diesem Programm mitgearbeitet hat, war diese Fassung „viele Jahre die einzige implementierte und funktionierende Semantik für Statcharts.“[3] Außerdem wollen wir beschreiben, wie Statecharts in der UML (Unified Modelling Language) dargestellt werden. Dabei berufen wir uns auf [Rum99] und [HK99a]. Weitere Ergänzungen schlägt Ian Horrocks in seinem Buch vor,[4] dessen Hauptanliegen allerdings nicht die Beschreibung von Statecharts, sondern eher eine Anleitung zur Benutzung derselben ist.

Es ist bereits aus dieser Aufzählung ersichtlich, dass es keine standardisierte Fassung gibt. Vielmehr hat jeder Autor seine eigene Variante, die sich von denen der anderen unterscheidet. Obwohl diese Unterschiede meist nur in Details liegen, kann dadurch gelegentlich unerwartetes Verhalten der Statecharts auftreten, wie es Leveson et al. in [Lev95] beschreiben. Die Autoren beziehen sich in diesem Fall auf eine von Pnueli und Shalev[5] beschriebene Semantik. Harel betont allerdings in [HN96], dass diese Probleme in der von ihm eingeführten Fassung (und der von ihm durchgeführten Erweiterungen) nicht auftauchen, sondern eher eine Schwäche dieser speziellen Variante sind.

In den folgenden Abschnitten werden wir nun einzeln auf die Grundelemente der Statecharts eingehen, und deren Varianten vorstellen. Es soll hierbei jedoch kein

[1] Alles fließt.
[2] vgl. [Har87b]
[3] vgl. [HN96], S. 294
[4] vgl. [Hor99a]
[5] vgl. [Pnu91]

Vergleich vorgenommen werden, der feststellt, welche Definition die bessere ist. Für eine derartige Gegenüberstellung sei auf [Beek94] verwiesen.[6]

2.2 Zustände

2.2.1 Grundformen der atomaren Zustände

Zustände können entweder atomare Zustände (*basic states*) oder zusammengesetzte Zustände sein, die eine Menge von Unterzuständen enthalten. Da zusammengesetzte Zustände alle Eigenschaften haben, die in atomaren Zuständen vorkommen, sollen letztere zunächst genauer beschrieben werden.

Atomare Zustände haben (bis auf den globalen Wurzelzustand) eingehende Verbindungen, durch die sie aktiviert werden. Wenn dies geschieht, können sie bereits ein entsprechendes Ereignis auslösen. Selbstverständlich gibt es meist auch ausgehende Verbindungen, über die der Zustand wieder verlassen werden kann; auch beim Verlassen des Zustands kann ein Ereignis ausgelöst werden.

Das Aktivieren und Deaktivieren eines Zustands wird auch als Betreten bzw. Verlassen des Zustands bezeichnet, eine Formulierung, die in dieser Bedeutung auch aus der Begriffswelt der Petri-Netze und Zustands-Übergangs-Diagramme bekannt ist.

Übliche Erweiterungen

Ausgehende Verbindungen sind nicht bei allen Zuständen notwendig – so wird beispielsweise der globale Wurzelzustand keine solchen besitzen. Dies heißt jedoch nicht in jedem Fall, dass ein solcher Zustand nicht mehr verlassen werden kann, da dies auch geschieht, wenn etwa ein Oberzustand verlassen wird. Darüber hinaus darf es aber auch Zustände geben, die tatsächlich nicht mehr verlassen werden können. Dies heißt jedoch nicht automatisch, dass sich das System beim Betreten eines solchen Zustands beendet; zunächst reagiert es lediglich nicht mehr auf weitere Eingaben und scheint daher stehen zu bleiben. Es ist jedoch naheliegend, eben dieses Beenden für einen oder mehrere Zustände zu spezifizieren. Diese Zustände werden dadurch zu terminalen Zuständen. Bemerkenswerterweise treten diese Zustände in Harels ursprünglicher Beschreibung [Har87b] nicht auf – sie werden jedoch in den meisten weiteren Schriften zusätzlich definiert, bei Harel selbst in Form der Terminal-Konnektoren,[7] bei Horrocks und in UML als Endzustände.[8] Insbesondere das aktive Beenden des Systems ist ein Argument dafür, diese Elemente tatsächlich zu den Zuständen (und nicht etwa den Konnektoren, die in Abschnitt 2.3 eingeführt werden) zu zählen.

Nicht alle Verbindungen brauchen vollständige Ereignis-Bedingung-Aktion-Gruppen zu besitzen. Ist bei einer ausgehenden Verbindung eines Zustands kein Ereignis (und eventuell auch keine Bedingung) angegeben, so wird dieser Übergang im nächsten Zeitschritt auf jeden Fall ausgeführt.[9] Ein solcher Zustand wird dann als transienter Zustand bezeichnet. Er ist den unten beschriebenen Konnektoren sehr ähnlich,

[6] Diese Auflistung wird von Harel und Namaad in [HN96] um das von ihnen verwendete Modell ergänzt.

[7] vgl. [HN96], S. 316

[8] vgl. [Hor99a], S. 64 f., oder [HK99a], S. 133 f., beziehungsweise [Rum99], S. 76

[9] vgl. jedoch die Beschreibung von auslöserfreien Übergängen der UML in Abschnitt 2.4.3

unterscheidet sich aber von ihnen dadurch, dass er immer noch ein echter Zustand ist, der aktiviert werden und Ereignisse auslösen kann.

Darüber hinaus gibt es die Möglichkeit, dem Zustand ein oder mehrere Ereignis-Bedingung-Aktion-Gruppen zuzuweisen, die in jedem Zeitschritt getestet und gegebenenfalls ausgeführt werden, in dem der Zustand aktiv bleibt. Dieses Verhalten nennt Harel „statische Reaktionen“[10] (*static reactions*). Es macht jedoch offensichtlich keinen Sinn, diese bei einem transienten Zustand anzugeben.

2.2.2 Zusammengesetzte Zustände

Zusammengesetzte Zustände ermöglichen die Tiefe (OR-Zustände) und Nebenläufigkeit (AND-Zustände), die Statecharts von Zustands-Übergangs-Diagrammen unterscheiden. Deshalb ist es nicht erstaunlich, dass sie in der Literatur weitgehend einheitlich in der folgenden Form beschrieben werden:

AND-**Zustand** Ein solcher Zustand ist in eine Menge von Unterzuständen aufgeteilt, die mit UND verbunden sind. Das heißt, wenn der Oberzustand aktiviert ist, so ist auch in jeder Partition ein Unterzustand aktiviert. Harel nennt diese Partitionen „orthogonale Komponenten“[11] (*orthogonal components*).

OR-**Zustand** Eigentlich wäre die Bezeichnung XOR-Zustand für diesen Typ von Zuständen genauer, denn sie ähneln einem Zustands-Übergangs-Diagramm, das aus mehreren Unterzuständen besteht, von denen immer genau ein einziger aktiviert ist, wenn der Oberzustand aktiviert ist.

Durch die Zuordnung der Zustände zu Oberzuständen entsteht eine neue Struktur, die generell mit dem Begriff der Zustandshierarchie bezeichnet wird.

Ein Unterschied zwischen atomaren und zusammengesetzten Zuständen ist der, dass der zusammengesetzte Oberzustand nicht direkt durch eine Verbindung erreichbar zu sein braucht. Der Zustand wird auch dann aktiviert, wenn ein Unterzustand betreten wird. Bei OR-Zuständen ist dies unproblematisch, da ja der zu aktivierende Unterzustand direkt angegeben ist; bei AND-Zuständen kann es vorkommen, dass bei dem Übergang eine oder mehrere der orthogonalen Komponenten nicht an dem Übergang teilhaben, aber selbstverständlich dennoch aktiviert werden müssen.

Für diesen Fall – und für den Fall, dass ein OR-Zustand direkt aktiviert wird – ist bei diesen üblicherweise einer ihrer Unterzustände als Defaulteinstieg markiert. Wird der OR-Zustand aktiviert, ohne dass ein spezieller Unterzustand betreten wird, so wird auch der so markierte Unterzustand aktiviert. Dies wird durch die Hierarchie von Zuständen so lange fortgesetzt, bis ein atomarer Zustand erreicht ist. Alternativ kann der Übergang (oder der Defaulteinstieg) auch auf einen Historykonnektor führen.[12]

2.2.3 Parametrisierte Zustände

Beim Modellieren von interaktiven Systemen kommt es gelegentlich vor, dass Zustände sich nur geringfügig voneinander unterscheiden oder gar mehrfach auftreten.

[10] vgl. [HN96], S. 297
[11] vgl. [HN96], S. 297, oder [HP98b], S. 61
[12] Die Funktionsweise dieses Konnektors wird in Abschnitt 2.3.2 genauer beschrieben

In [Har87b] schlägt Harel für solche Fälle das Konzept der parametrisierten Zustände vor.[13] Ein solcher Zustand wird im Statechart nur ein einziges Mal genau ausgearbeitet, doch beim laufenden System kann er mehrere Entsprechungen haben.

Harel unterscheidet zwei Formen der Parametrisierung, die OR- bzw. AND-Parametrisierung. Mit dieser Bezeichnung wird angegeben, wie die einzelnen Instanzen des parametrisierten Zustands zueinander stehen: in einer Gruppe von Instanzen eines OR-parametrisierten Zustands wird immer nur genau einer aktiv sein (man denke etwa an einen Zähler), während eine AND-Parametrisierung für orthogonale, voneinander unabhängige Vorgänge verwendet wird, wie zum Beispiel das Verwalten unterschiedlicher Anrufe in einer Telefonzentrale.

Horrocks schlägt eine leicht modifizierte Variante von parametrisierten Zuständen vor, die er in einem Beispiel zum Anzeigen unterschiedlicher Fehlermeldungen verwendet.[14] Dabei ist lediglich die Funktionsweise des Fehler-Dialogs im Zustand selbst modelliert, der Text der Meldung wird diesem beim Betreten übergeben. Dies kann beispielsweise durch eine Textvariable geschehen. Horrocks Fassung unterscheidet sich von beiden von Harel beschriebenen Formen, da weder immer genau eine Instanz des parametrisierten Zustands aktiv ist, noch mehrere Instanzen nebeneinander existieren können.

Die Idee der Parametrisierung von Zuständen wird darüber hinaus in der Literatur so gut wie nicht aufgegriffen, und es gibt auch keine einheitliche Notation dafür. In Statemate gibt es stattdessen Generische Charts, die eine ähnliche Funktion erfüllen wie AND-parametrisierte Zustände, da sie mehrfach instanziiert werden können.[15] Diese Charts sind – anders als die ursprünglichen parametrisierten Zustände – aus dem zentralen Statechart ausgekoppelt und kommunizieren mit ihm nur noch über Parameter. Insbesondere sind sie damit auch vom eigentlichen Namensraum und der Sichtbarkeit von Ereignissen getrennt. Auf diese Weise werden einige Probleme gelöst, die bei der AND-Parametrisierung auftreten; auf diese Schwierigkeiten soll in Abschnitt 4.2.1 genauer eingegangen werden.

2.2.4 Denkbare Erweiterungen des Zustandsbegriffs

In seiner ersten Beschreibung der Statecharts verfolgt Harel nicht den in späteren Schriften üblichen Ansatz, die Zustände streng hierarchisch zu ordnen, sondern lässt zu, dass sich zwei Zustände, die im Prinzip auf einer Hierarchieebene liegen, teilweise überschneiden (siehe Abbildung 2.1). Dies hat den Vorteil, dass man Zustände gruppieren kann, die nicht notwendigerweise demselben übergeordneten Zustand angehören.[16]

Obwohl diese Möglichkeit in einigen Fällen die Zahl der notwendigen Zustände und Übergänge verringert, ist es keine echte Erweiterung der Ausdrucksfähigkeit. Anstelle des Bereichs der Überschneidung lässt sich ein neuer Zustand auf derselben Hierarchieebene einführen, so dass drei schnittfreie Zustände entstehen, die das System äquivalent beschreiben (in Abbildung 2.2 wird dies dargestellt). Allerdings werden dadurch einige Bedingungen umfangreicher; so muss in dem dargestellten Statechart

[13] vgl. [Har87b], S. 258 ff.
[14] vgl. [Hor99a], S. 89 ff.
[15] vgl. [HP98b], S. 187 ff.
[16] vgl. [Har88], S. 527

beispielweise die Abfrage, ob Zustand `A` aktiviert ist, auf die Zustände `A` und `G` erweitert werden.

Diese Ergänzung der Statecharts wird in der Literatur nicht weiter verfolgt und auch in Statemate nicht implementiert.

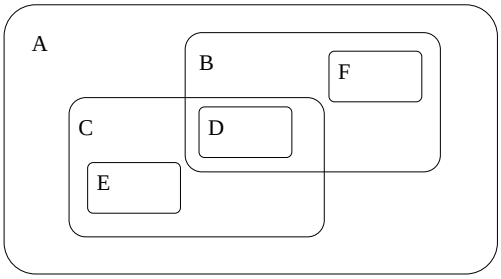


Abbildung 2.1: überlappende Zustände

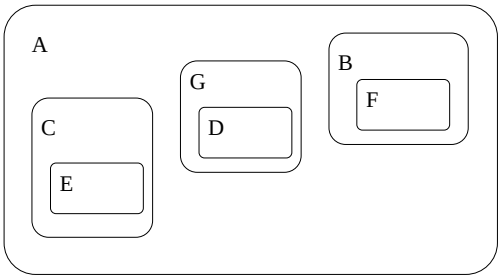


Abbildung 2.2: Auflösung der überlappenden Zustände

Als weitere denkbare Erweiterung beschreibt Harel sogenannte unsichere Zustände („Not-Quite-Sure“-Blobs).[17] Er will damit die Unsicherheit des Statechart-Architekten formulierbar machen, falls nicht klar ist, ob in einem bestimmten Bereich des Statecharts Zustände vorhanden sind. Er kennzeichnet solche Zustände wie in Abbildung 2.3 und schreibt dazu:[18]

> „[Diese Darstellung] erklärt unsere Unsicherheit, ob `A - B` leer ist oder nicht, und stellt außerdem fest, dass die Differenz, falls sie nicht leer ist, mit `E` bezeichnet wird, und mit `F` durch die Beziehung `R` verbunden ist.“

Da jedoch auch bei den üblichen Zuständen im Statechart nicht zwingend vorausgesetzt wird, dass sie erreichbar sind (und es sie daher im obigen Sinne „gibt“), kann dies ebenso ohne die Einführung unsicherer Zustände modelliert werden. Auch diese Idee wird daher weder in weiterer Literatur aufgegriffen noch in Statemate implementiert.

Schließlich spielt Harel noch mit dem Gedanken, dass es Zustände geben könnte, die betreten werden dürfen, ohne dass dadurch ein Oberzustand aktiviert wird. Harel

[17] vgl. [Har88], S. 526
[18] vgl. [Har88], S. 526

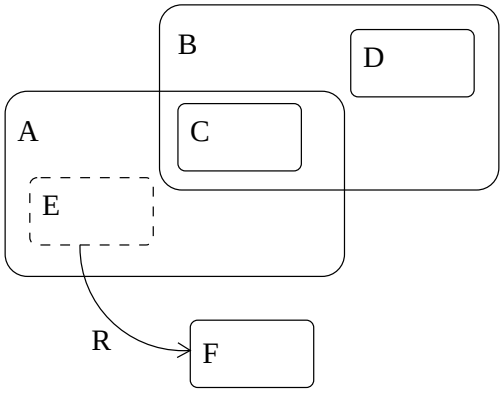


Abbildung 2.3: unsichere Zustände

gibt ein Beispiel, bei dem er die oben beschriebenen überlappenden Zustände verwendet:[19] So ist es zum Beispiel denkbar, dass ein Zustand in zwei sich nur teilweise überschneidenden Oberzuständen liegt, und dass beim Betreten dieses Zustands nur einer der beiden Oberzustände aktiviert werden soll. Bezogen auf Abbildung 2.1 hieße dies, dass etwa Zustand `D` betreten wird und Zustand `B` dadurch ebenfalls aktiviert wird, während Zustand `C` deaktiviert bleibt. Dieses Konzept steht jedoch im Widerspruch zur sonst verfolgten strengen Hierarchie. Vermutlich liegt darin begründet, dass in der Literatur nicht näher darauf eingegangen wird.

2.3 Konnektoren

2.3.1 Grundformen der Konnektoren

Zu Konnektoren lässt sich zunächst bemerken, dass dieser Begriff in [Har87b] und [Har88] noch nicht auftaucht. Auch Horrocks verzichtet in [Hor99a] auf diese Klasse von Elementen. Allerdings treten in jeder Beschreibung von Statecharts Elemente auf, die keine vollwertigen Zustände sind, aber dennoch meistens, mangels eines besseren Begriffs, als Zustände[20] bezeichnet werden.

In Statemate werden diese Elemente als Konnektoren klassifiziert, und in späteren Arbeiten übernimmt Harel diesen Begiff. Wir wollen uns hier dieser Bezeichnungsweise anschließen, da sie die Unterscheidung zu den eigentlichen Zuständen leichter macht.

Konnektoren ähneln transienten atomaren Zuständen in der bereits in Abschnitt 2.2.1 beschriebenen Form zu großen Teilen; der Unterschied besteht allerdings darin, dass Konnektoren nicht wie Zustände aktiviert werden können. In Statechart-Modellen mit diskreter Zeit verzögern sie die Reaktion des Systems daher nicht um einen Zeitschritt; generell können sie insbesondere keine Ereignisse oder Aktionen auslösen und sind damit eher Teil der Verbindung.

[19] vgl. [Har88], S. 527 f.
[20] in UML als *Pseudozustände*, vgl. zum Beispiel [HK99a], S. 133, oder [Rum99], S. 397

Konnektoren in Statemate

In Statemate werden sieben Formen von Konnektoren unterschieden, die auch in den Arbeiten, die sich auf dieses Programm beziehen, beschrieben werden.[21]

Die ersten beiden sind die AND-Konnektoren Gabelung (*fork*) und Vereinigung (*join*). Diese beiden Konnektoren modellieren den Fall, dass mehrere Zustände durch einen einzigen Übergang gleichzeitig betreten oder verlassen werden. Ein solcher Übergang ist nur dann möglich, wenn die Bedingungen und Ereignisse, die an allen Teilen dieser Verbindungen stehen, eingetreten und alle Startzustände aktiv sind. Nach einem solchen Übergang werden alle Startzustände deaktiviert und alle Zielzustände aktiviert. Abbildung 2.4 stellt zwei einfache Beispiele für derartige Übergänge dar.

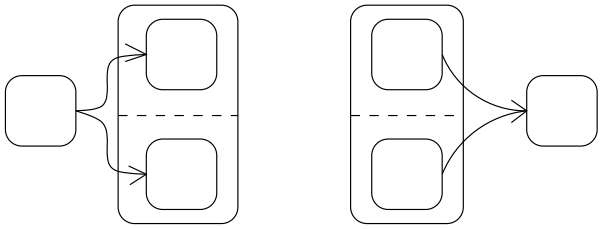


Abbildung 2.4: Übergänge mit mehr als einem End- bzw. Startzustand

Diese Konnektoren sind insbesondere im Zusammenhang mit AND-Zuständen zu sehen. Obwohl es denkbar ist, dass ein AND-Zustand durch einen Übergang verlassen werden kann, der von einem Unterzustand nur einer einzigen orthogonalen Komponente ausgeht, muss die Möglichkeit gegeben sein, diese Bedingung auf alle orthogonalen Komponenten erweitern zu können. Auf diese Weise entsteht ein Übergang, der in mehreren Unterzuständen des AND-Zustands beginnt und an einem einzigen Punkt aufhört.

Andererseits kann es gewünscht sein, beim Betreten eines AND-Zustands sämtliche Unterzustände der orthogonalen Komponenten angeben zu können, die aktiviert werden sollen. Dies wird durch einen Übergang dargestellt, der an einem Punkt beginnt und in mehreren Unterzuständen des AND-Zustands endet.

Die zweite Gruppe der Konnektoren wird durch die OR-Konnektoren Bedingung (*condition*), Verteiler (*switch*) und Verzweigung (*junction*) gebildet. Diese Konnektoren haben gemein, dass sie mehrere eingehende Verbindungen akzeptieren und darüber hinaus mehrere ausgehende Verbindungen anbieten, von denen jedoch nur genau eine durch die Kombination von zugeordnetem Ereignis und Bedinung erlaubt sein darf, um das Modell deterministisch zu halten. Wären mehrere Übergänge möglich, so ist zunächst nicht bestimmt, welcher von diesen wirklich ausgeführt wird. Auf die hier auftretende Frage des Determinismus wird in Abschnitt 2.4 noch genauer eingegangen.

Letztendlich gibt es in Statemate noch einen weiteren Typ von Konnektor, den sogenannten Diagramm-Konnektor, der lediglich zwei unterschiedliche Punkte eines Diagramms identifiziert und es dadurch möglich macht, lange Pfeile quer durch das

[21] vgl. [HN96], S. 302, oder [HP98b], S. 65 ff.

Diagramm zu vermeiden. Da die Semantik hierdurch nicht erweitert wird, sollte man die einander entsprechenden Paare von Diagramm-Konnektoren laut Harel so betrachten, als seien sie „eigentlich ein einziger OR-Konnektor, der einfach an mehreren Stellen auftaucht.“[22]

Entsprechungen der Konnektoren in UML

Der Begriff des Konnektors wird in der UML nicht benutzt. Hitz und Kappel sprechen stattdessen von Pseudozuständen, die sie als „reine notationstechnische Krücke“ bezeichnen – „keine Zustände im eigentlichen Sinn.“[23] Wir wollen sie daher hier wie Konnektoren aufführen.

Zunächst sei erwähnt, dass UML die beiden Formen der AND-Konnektoren durch sogenannte Synchronisationsbalken anbietet. Obwohl die Darstellungsweise sich von der hier beschriebenen unterscheidet, soll darauf nicht genauer eingegangen werden, da die Semantik exakt dieselbe ist.[24]

UML bietet zwei Verbindungsstellen, die den OR-Konnektoren entsprechen, die statische Verzweigung (*static branch point*) und die dynamische Verzweigung (*dynamic choice point*).[25] Beide Typen dürfen mehrere ausgehende Verbindungen haben, die allerdings nicht durch weitere Ereignisse sondern lediglich durch Bedingungen beschränkt sein dürfen. Bei statischen Verzweigungen werden sämtliche Bedingungen vor dem Übergang ausgewertet, bei dynamischen Verzweigungen werden zunächst die Aktionen der eingehenden Verbindungen ausgeführt und dann die entsprechende ausgehende Verbindung gewählt.

Um die Bedeutung dieser Verbindungsstellen genauer zu verstehen, muss erwähnt werden, dass UML von einem kontinuierlichen Zeitmodell[26] ausgeht; es kommt daher durchaus auf die Reihenfolge an, in der einzelne Ereignisse entlang eines Übergangs auftreten. Damit wird es wichtig festzustellen, zu welchem Zeitpunkt etwa das Verlassen eines Zustands ein Ereignis auslöst. Mit Hilfe der statischen Verzweigungen lässt sich diese Reihenfolge ändern, wie in Abbildung 2.5 dargestellt wird: Auf das Ereignis `f` werden (in dieser Reihenfolge) die Ereignisse bzw. Aktionen `exit(C)`, `x`, `exit(A)`, `y`, `entered(B)`, `z` und `entered(D)` generiert.[27]

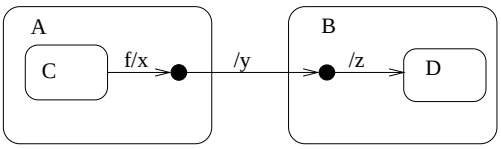


Abbildung 2.5: Reihenfolge von Aktionen bei statischen Verzweigungen

[22] vgl. [HN96], S. 302
[23] vgl. [HK99a], S. 133
[24] Um dem Leser einen Eindruck dieser Darstellung zu geben benutzen wir Synchronisationsbalken jedoch im Beispiel der Abbildung 2.6.
[25] vgl. [HK99a], S. 146 f.
[26] vgl. Abschnitt 2.8
[27] nach [Rum99]

Bei dynamischen Verzweigungen wird die ausgehende Verbindung gewählt, nachdem die Aktionen der eingehenden Verbindungen ausgeführt wurden. Hitz und Kappel gehen dabei nicht darauf ein, was passiert, wenn durch diese Aktion keine der Bedingungen der ausgehenden Verbindungen mehr erfüllt ist; dies lässt sich jedoch einfach sicherstellen, wenn man vorschreibt, dass genau eine der Bedingungen die spezielle Bedingung `else` sein muss, die immer dann wahr ist, wenn alle anderen angegebenen Bedingungen falsch sind.

Statische Verzweigungen mit mehreren ausgehenden Verbindungen sind trotz ihres Namens nicht die beste Wahl für Verzweigungen aufgrund von Abfragen. Derartige Übergänge sollten in UML eher durch Entscheidungsknoten modelliert werden. Diese Konnektoren besitzen dieselbe semantische Bedeutung wie statische Verzweigungen, werden jedoch grafisch durch eine Raute dargestellt. Dadurch ist leichter festzustellen, an welchen Punkten des Statecharts derartige Verzweigungen stattfinden.

Durch den kontinuierlichen Zeitverlauf ist es in UML nicht möglich, sicherzustellen, dass ein Zustand auf die Ausgabe eines anderen Zustands wartet (Produzent-Konsument-Beziehung). Um dieses Verhalten modellieren zu können, wurde ein weiterer Pseudozustand eingeführt, der Synch-Zustand.[28] Er funktioniert wie ein Speicher (wahlweise mit beschränkter oder unbeschränkter Kapazität), der zählt, wie oft die eingehenden Verbindungen aktiviert wurden. Ausgehende Übergänge sind genau dann möglich, wenn der Speicher nicht leer ist; jedes Mal, wenn ein solcher Übergang ausgeführt wird, wird der Speicher um eins heruntergezählt. Dieser Konnektor ist insbesondere dann sinnvoll, wenn er einer von mehreren Startzuständen einer Verbindung mit einem Synchronisationsbalken ist. Ein Beispiel ist in Abbildung 2.6 dargestellt, die aus Hitz und Kappel übernommen wurde.[29] Der Übergang im Konsumenten ist hierbei nur dann möglich, wenn vorher im Produzenten mindestens ein Übergang ausgeführt wurde, der noch im Synch-Zustand (hier mit unbegrenzter Kapazität, gekennzeichnet durch den Stern) vorgemerkt ist.

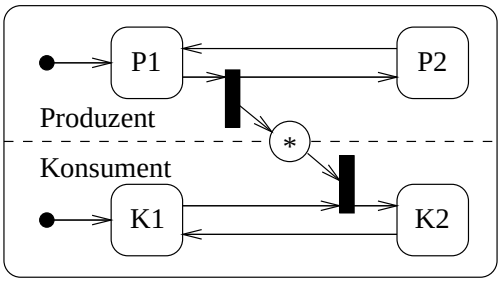


Abbildung 2.6: Produzent und Konsument sind über einen Synch-Zustand verbunden

2.3.2 Der Historykonnektor

Der Historykonnektor tritt in sämtlichen betrachteten Schriften in praktisch einheitlicher Form auf. Allerdings wird er meist in der Form eines History-Einstiegs beschrie-

[28] vgl. [HK99a], S. 132, oder [Rum99], S. 463 ff. *Anmerkung: dies ist kein Tippfehler!*
[29] vgl. [HK99a], S. 148

ben.[30] Dieser Einstieg ermöglicht es dem Architekten des Systems, einen Zustand vorübergehend zu verlassen (etwa um einen Interrupt abzuarbeiten) und danach zur vorherigen Situation des Zustands zurückzukehren. Durch die Einführung von Konnektoren in Statemate ist diese Funktionalität sinnvollerweise in diese Gruppe von Elementen übertragen worden.

Eigentlich ist jedoch die Bezeichnung als Einstieg ungenau. In den meisten Beschreibungen erhält ein Zustand erst dann eine *History*, wenn er zum ersten Mal betreten wurde. Daher kann es passieren, dass noch keine History vorhanden ist, wenn der Historyeinstieg benutzt werden soll. Das Verhalten des Systems wird in diesem Fall durch den Begriff Einstieg nicht intuitiv erklärt. Wird stattdessen ein Konnektor benutzt, ist eine derartige Beschreibung möglich, wie im Folgenden beschrieben wird. Historykonnektoren brauchen (im Gegensatz zu AND- und OR-Konnektoren) keine ausgehenden Verbindungen zu besitzen, dürfen dies aber – was bei einem Historyeinstieg im wörtlichen Sinn nicht möglich ist. Das Verhalten ist das folgende: Ist eine History vorhanden, so wird der auf diese Weise gespeicherte Zustand wieder hergestellt. Ist aber keine History vorhanden (weil der Zustand zum ersten Mal betreten wird oder die History explizit gelöscht wurde), so wird eine vom Historykonnektor ausgehende Verbindung genutzt. Ist keine solche vorhanden (oder mit einer Bedingung versehen, die nicht erfüllt ist), so wird der Defaulteinstieg verwendet.

Die Möglichkeit, eine solche Verbindung anzugeben, ist besonders interessant, wenn der Defaulteinstieg auf einen Historykonnektor verweist. Wird ein solcher Zustand zum ersten Mal betreten und dabei der Defaulteinstieg benutzt, so würde das System in eine Schleife geraten: der Defaulteinstieg will den Historyeinstieg benutzen, da aber noch keine History vorhanden ist, wird wieder der Defaulteinstieg betrachtet. Eine vom Historykonnektor ausgehende Verbindung löst dieses Problem.

Außerdem gibt es in der Literatur zwei Arten des Historykonnektors, die sich darin unterscheiden, ob sie sich lediglich auf den Zustand beziehen, in dem sie sich befinden, oder auch auf die jeweils aktiven Unterzustände. Die zweite Variante wird meist als *deep history* bezeichnet. Während der einfache Historykonnektor lediglich die History des ersten Zustand betrachtet, und in allen darunterliegenden Zuständen die Defaulteinstiege nutzt, stellt der Deep-Historykonnektor auch in allen Unterzuständen die zuletzt aktivierte Situation wieder her.

Damit gibt es vier Varianten des Historykonnektors, die sich durch Reichweite und Verhalten bei fehlender History unterscheiden.

2.4 Verbindungen

Verbindungen dienen dazu, den Folgezustand eines aktivierten Zustands zu bestimmen, also den Übergang von einem aktivierten Zustand auf den nächsten. Generell besitzt eine Verbindung drei Attribute, nämlich ein Ereignis, eine Bedingung und eine Aktion. Diese Gruppe braucht jedoch nicht in jedem Fall vollständig zu sein. So hat zum Beispiel eine von einem transienten Zustand ausgehende Verbindung weder ein Ereignis noch eine Bedingung, da dieser Übergang in jedem Fall ausgeführt werden soll. Andererseits braucht auch nicht jede Verbindung spezielle Aktionen auslösen.

[30] vgl. beispielsweise [Har87b], S. 238, oder [Hor99a], S. 65 f

Verbindungen ohne auslösendes Ereignis haben in der UML eine weitere Bedeutung, auf die in Abschnitt 2.4.3 genauer eingegangen werden soll.

Es ist nicht ausgeschlossen, dass zwischen einem Ausgangs- und einem Zielzustand mehrere parallele Verbindungen bestehen, die unterschiedliche Ereignisse oder Bedingungen tragen und unterschiedliche Aktionen auslösen können.

Zur Darstellung sei einleitend erwähnt, dass Harel in seinen Abbildungen auch Verbindungen mit Namen versieht. Diese Bezeichner dienen jedoch nur zur leichteren Bezugnahme und sind kein Teil der Statechart-Syntax. Wir wollen uns dieser Möglichkeit in gleicher Weise bedienen.

2.4.1 Zusammengesetzte Übergänge

Generell beginnen und enden Verbindungen bei Zuständen, die bei Ausführen des Übergangs verlassen beziehungsweise betreten werden. Bei genauerer Betrachtung kann eine solche Verbindung allerdings aus einzelnen Teilen bestehen, die zum Beispiel mehrere Konnektoren aneinanderreihen. Auf diese Weise entsteht ein zusammengesetzter Übergang (*compound transition*).

Wird das diskrete Zeitmodell benutzt, so entspricht eine derartige Konstruktion einem einzigen Übergang mit mehreren Einträgen, wie es Harel und Namaad in [HN96] erwähnen:[31]

> „Der Übergang t_1 in Abbildung 2.7(a) kann nicht ausgeführt werden, ohne dass auch t_2 ausgeführt wird. [...] Sein Auslöser ist *ev1 & ev2* und seine Aktion ist *act1; act2*. Daher ist Abbildung 2.7(a) äquivalent zu Abbildung 2.7(b)."

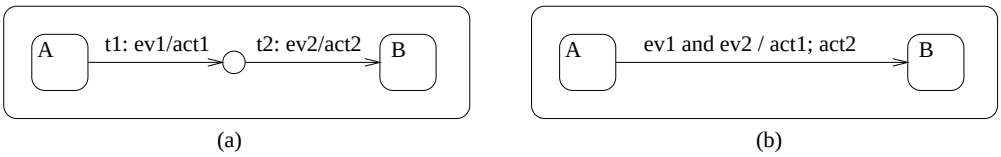


Abbildung 2.7: äquivalente zusammengesetzte Übergänge

Im kontinuierlichen Zeitmodell kann ein zusammengesetzter Übergang allerdings nicht äquivalent ersetzt werden, da es auf die Reihenfolge der ausgelösten Ereignisse ankommen kann, wie bereits in Abbildung 2.5 im Zusammenhang mit statischen Verzweigungen in UML gezeigt wurde.

Ein zusammengesetzter Übergang liegt selbstverständlich auch vor, wenn ein Konnektor benutzt wird, von dem mehrere Verbindungen ausgehen, wie es bei OR-Konnektoren üblich ist. In diesem Fall ist der Übergang auf den Konnektor nur dann möglich, wenn mindestens eine der ausgehenden Verbindungen möglich ist.

Für derartige Konstruktionen werden in der Literatur verschiedene Zusatzbedingungen gefordert. Für Statemate und UML müssen die Bedingungen der ausgehenden Äste sich gegenseitig ausschließen, um Nicht-Determinismus zu verhindern,[32] in UML

[31] vgl. [HN96], S. 303

[32] vgl. [HP98b], S. 65 f., beziehungsweise [HK99a], S. 147, oder [Rum99], S. 73

dürfen die ausgehenden Verbindungen darüberhinaus nicht mit weiteren Ereignissen versehen sein.[33]

In [HN96] erzeugen die Autoren noch auf eine andere Weise zusammengesetzte Übergänge: bei ihnen ist es möglich, auch bei den Verbindungen, die von einem Defaulteinstiegs-Konnektor ausgehen, weitere Ereignisse und Bedingungen anzugeben. Eine Konstruktion wie in Abbildung 2.8 führt dann dazu, dass der Übergang t_1 nur dann erlaubt ist, wenn auch t_2 ausgeführt werden kann.[34]

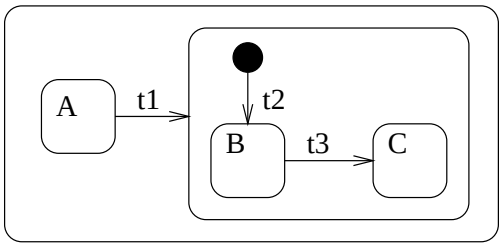


Abbildung 2.8: Defaulteinstieg mit Bedingung

Der Umgang mit zusammengesetzten Verbindungen ist jedoch nicht unproblematisch. So schildern Harel und Namaad in diesem Zusammenhang ein Beispiel, bei dem der Statemate-Simulator sich anders verhält als der von Statemate erzeugte Quelltext.[35] Dieses Beispiel wird in Abbildung 2.9 dargestellt:

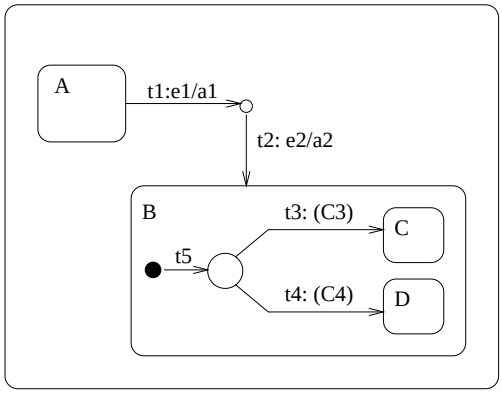


Abbildung 2.9: Unterschiede bei Simulation und Quelltext

Wenn die Ereignisse *e1* und *e2* eingetreten, die Bedingungen *C3* und *C4* aber nicht erfüllt sind, so ist der Übergang von `A` auf `B` nicht möglich. Der Simulator wird dies erkennen, und den Übergang nicht ausführen. Zum erzeugten Quelltext schreiben die Autoren:[36]

[33] vgl. [HK99a], S. 146
[34] vgl. [HN96], S. 304
[35] vgl. [HN96], S. 303
[36] vgl. [HN96], S. 303

> „[Er] wird den Anfangsteil des Übergangs ausführen, und erst nach Betreten von B feststellen, dass keine atomaren Zustände betreten werden können; an dieser Stelle wird er keinen *Rollback* auf A durchführen. Es ist theoretisch möglich, Code zu erzeugen, der unter diesen Umständen den vorherigen Zustand wieder herstellt, doch wir konnten keinen effizienten Weg finden, dies zu tun. Dieser Punkt scheint noch weitere Forschung zu verlangen."

2.4.2 Determinismus und Prioritäten

Wichtig ist für jeden Zustand, der mehrere ausgehende Verbindungen besitzt, dass nur eine davon wirklich als Übergang ausgeführt wird. Dies kann entweder dadurch sichergestellt werden, dass sie mit unterschiedlichen Ereignissen versehen sind, oder dadurch, dass die Bedingungen, die den Verbindungen zugeordnet sind, sich gegenseitig ausschließen. Es kann jedoch Situationen geben, in denen diese Möglichkeiten nicht ausreichen.

Deshalb wird in einigen Schriften vorgeschlagen, Verbindungen mit Prioritäten zu versehen, die im Falle mehrerer möglicher Übergänge eine deterministische Entscheidung herbeiführen. Diese Prioritäten sind meist ganze Zahlen, beispielsweise in [HN96]. Horrocks geht sogar noch einen Schritt weiter und empfiehlt die Benutzung von Prioritäten ausdrücklich, um Bedingungen zu vereinfachen. Er führt als Beispiel das folgende Detail eines Taschenrechners an, bei dem es um die Division durch Null geht:[37] Es wurden bereits zwei Operanden (der zweite davon `0`) und der Bruchstrich als Operator eingegeben, und nun wird die Taste `=` gedrückt. Die beiden Statecharts (a) und (b) in Abbildung 2.10 modellieren in äquivalenter Weise die Reaktionen auf den Fehler, doch ist die Fassung (b) aufgrund der Prioritäten (in eckigen Klammern) leichter zu überblicken.

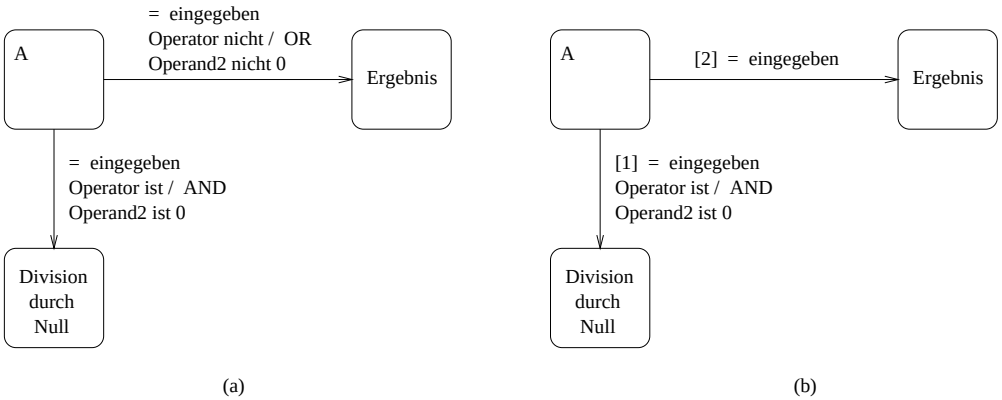


Abbildung 2.10: Prioritäten an Übergängen

Nichtdeterminismus kann auch über mehrere Hierarchieebenen hinweg auftreten. Dies ist zum Beispiel der Fall, wenn ein bestimmter Zustand und ein ihm untergeordneter Zustand zur gleichen Zeit verlassen werden sollen. In Statemate wurde für diese

[37] vgl. [Hor99a], S. 106 f.

Fälle der Begriff der strukturellen Priorität (*structural priority*) eingeführt.[38] Diese berechnet sich aus der Lage der Verbindungen innerhalb des Statecharts, das heißt aus der Tiefe der Start- und Zielzustände in der Hierarchie. Harel und Namaad beschreiben zwei Wege zur Berechnung dieser Priorität, da sich verschiedene Versionen von Statemate in diesem Punkt unterscheiden.

In der Fassung der ersten Versionen wurde der (bezüglich der Hierarchie) unterste Zustand betrachtet, der sämtliche Start- und Zielzustände enthielt (Harel und Namaad nennen diesen Zustand *scope*); für neuere Versionen werden hauptsächlich die Startzustände dazu verwendet. Die Formel, die in [HN96] in diesem Zusammenhang beschrieben wird, lässt sich wie folgt zusammenfassen: Hat ein Übergang nur einen Startzustand, so wird dessen Hierarchietiefe als strukturelle Priorität verwendet, ansonsten die des höchsten AND-Zustands, der durch den Übergang verlassen wird.

Die Unterschiede dieser beiden Berechnungsarten zeigen sich etwa in folgendem Beispiel, das aus [HN96] entnommen wurde (siehe Abbildung 2.11). In der ersten Form erhalten die Übergänge t_1 und t_3 dieselbe Priorität (eine größere als die von t_2) da beide in nur in Zustand A ganz enthalten sind. Ist also Zustand C (und damit auch Zustand B) aktiv und die Übergänge t_1 und t_3 möglich, so ist das Verhalten des Statecharts nicht deterministisch, solange keine weiteren expliziten Prioritäten angegeben werden. Die zweite Fassung gibt t_1 und t_2 dieselbe Priorität (eine kleinere als die von t_3), da sie beide bei C beginnen. In diesem Fall ist das System auch dann noch deterministisch, wenn Zustand C aktiv ist und alle drei Übergänge möglich sind.

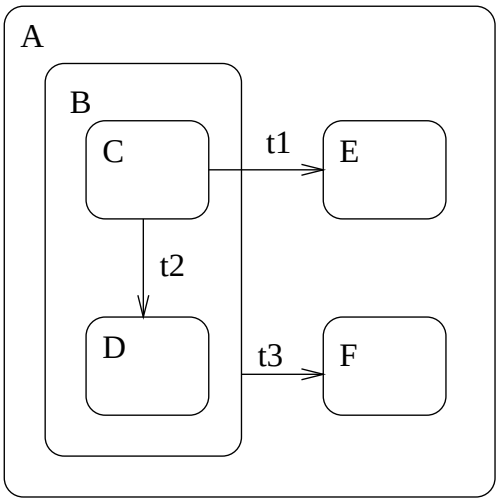


Abbildung 2.11: Strukturelle Priorität

2.4.3 Denkbare Erweiterungen des Übergangsbegriffs

Besondere Erwähnung soll hier eine Konstruktion der UML finden, die auslöserfreier Übergang (*triggerless transition*) genannt wird. Obwohl eine solche Verbindung, wenn sie von einem atomaren Zustand oder einem Konnektor ausgeht, zu jedem Zeit-

[38] vgl. [HN96], S. 328 ff.

punkt ausgeführt werden kann, besitzt sie im Fall von komplexen Zuständen eine andere Bedeutung. Hierbei wird sie durch die Tatsache ausgelöst, dass (bei AND-Zuständen in allen Partitionen) ein Endzustand erreicht wurde.[39] Das auslösende Moment wird hierbei nicht als Ereignis angesehen, da es nur für einen derartigen Übergang sichtbar ist.

Ebenso wie für den Zustandsbegriff gibt Harel in [Har88] Varianten der Verbindungen an. Anstelle den grafischen Verbindungen lediglich zuzugestehen, dass sie Anfangs- und Endzustand miteinander verbinden, könnte man auch den Weg betrachten, den sie dabei zurücklegen. Insbesondere wäre es dadurch möglich, einen Zustand während eines Übergangs mehrfach zu verlassen und wieder zu betreten. Dies ist jedoch nur dann sinnvoll, wenn man ein kontinuierliches Zeitmodell zugrunde legt, da ansonsten die Ereignisse ohnehin nur im nächsten Zeitschritt sichtbar sind, und es hierbei keinen Unterschied macht, ob ein Ereignis einmal oder mehrfach aufgetreten ist.

Er regt weiterhin an, dass man es Verbindungen ermöglichen könnte, grafische Grenzen eines Zustands, die überschritten werden müssen, um den geplanten Weg der Verbindung einzuzeichnen, zu ignorieren. Dies führt jedoch wieder auf Zustände, deren Aktivierung oder Deaktivierung von ihren Oberzuständen unabhängig ist, wie sie bereits in Abschnitt 2.2.4 angesprochen wurden.

In [Har87b] schlägt er vor, stochastischen Nichtdeterminismus in die Statecharts einzubauen, indem etwa Wahrscheinlichkeiten für verschiedene Übergänge angegeben werden. Diese Erweiterung nennt er Markov-Charts, da sie Markov-Ketten in ähnlicher Weise verallgemeinern wie Statecharts die Zustands-Übergangs-Diagramme. Es wird jedoch in der Literatur nicht weiter darauf eingegangen, da die modellierten Systeme meist deterministisch bleiben sollen.

2.5 Ereignisse

Mit Ereignissen sind Signale gemeint, die gesetzt werden, um das System zu beeinflussen. Diese Einflussnahme geschieht dadurch, dass durch die aufgetretenen Ereignisse Übergänge ausgelöst werden können. Je nach Herkunft sind dabei interne und externe Ereignisse zu unterscheiden.

Als interne Ereignisse werden diejenigen bezeichnet, die das Statechart selbst erzeugt. Dazu gehören etwa solche, die automatisch beim Betreten oder Verlassen eines Zustands auftreten, aber auch diejenigen, die im Aktionsteil eines Übergangs ausgelöst werden. Bei den externen Ereignissen, die für das Statechart gewissermaßen von außen auftauchen, gibt es andererseits so viele Möglichkeiten, dass diese in der Literatur nicht genauer untersucht werden.[40]

2.5.1 Sichtbarkeitsdauer

Allen Ereignissen ist gemein, dass sie nicht lange sichtbar sind. Was darunter jedoch genau zu verstehen ist, hängt vom gewählten Zeitmodell ab. Basiert das Statechart auf

[39] vgl. [Rum99], S. 484

[40] Im Zusammenhang mit *SCed* sei jedoch auf die Überlegungen zur Programmierung solcher Ereignisse in Abschnitt 3.3.2 hingewiesen.

einer diskreten Zeit, so sind Ereignisse üblicherweise nur einen Zeitschritt lang vorhanden. Sie können in diesem ihrer Erzeugung folgenden Zeitschritt Übergänge auslösen (sofern die zugehörigen Bedingungen das zulassen) und werden dann wieder gelöscht. Einige Autoren definieren die Sichtbarkeit von Ereignissen derart, dass sie schon in dem Zeitschritt, in dem sie erzeugt wurden, weitere Übergänge auslösen können.[41]

In der UML, die wie bereits erwähnt von einer kontinuierlichen Zeit ausgeht, ist die Definition der Existenzdauer eines Ereignisses nicht so präzise. Hitz und Kappel schreiben dazu:[42] „Tritt ein Ereignis ein, dem im aktuellen Zustand keine Transition zugeordnet ist, wird es ignoriert – es »geht verloren«," beziehungsweise im Zusammenhang mit einer nicht erfüllten Bedingung:[43]

> „Das in diesem Fall »nicht konsumierte« Ereignis geht verloren, das heißt, selbst wenn die Bedingung nach Eintritt des Ereignisses irgendwann erfüllt wird, kann die Transition ohne neuerliches Ereignis nicht durchgeführt werden."

Um auf diese Weise keine Ereignisse zu verlieren, die aus Sicht des Systems etwas zu früh eingetreten sind, gibt es die Möglichkeit, ein Ereignis in einem Zustand aufzuschieben (beziehungsweise als Ereignis zu markieren, das für diesen Zustand aufgeschoben werden soll; meist wird dies als *deferred* bezeichnet). Wenn ein solches Ereignis erzeugt wird aber momentan keinen Übergang auslöst, so bleibt es erhalten, bis eine von zwei Möglichkeiten auftritt: Entweder löst es doch einen Übergang aus, weil zum Beispiel nun eine Bedingung erfüllt ist, die den Übergang zuvor blockiert hatte. Oder das System geht in einen Zustand über, für den das Ereignis nicht mehr aufzuschieben ist; in diesem Fall wird es wie ein normales Ereignis behandelt und kann gegebenenfalls verlorengehen.

2.5.2 Die Rahmenbedingung

Ereignisse sind zunächst für das gesamte System sichtbar und in diesem Sinne global. Gewissermaßen wird also jeder aktive Zustand nach seiner Reaktion auf jedes eintretende Ereignis gefragt. Es gibt dabei die folgenden Möglichkeiten.

Einerseits kann eine ausgehende Verbindung des Zustands durch das eingetretene Ereignis ausgelöst werden. In diesem Fall hängt es von der dieser Verbindung zugeordneten Bedingung ab, ob tatsächlich ein Übergang stattfindet. Je nach dem wird der Zustand dann deaktiviert oder bleibt aktiviert. Andererseits kann es vorkommen, dass zu einem Ereignis keine entsprechende Verbindung gefunden wird, die von dem aktiven Zustand ausgeht. In diesem Fall hängt die Reaktion des Statecharts von der gewählten Rahmenbedingung ab.

Bei der schwachen Rahmenbedingung wird davon ausgegangen, dass ein Zustand, dessen ausgehende Verbindungen das Ereignis nicht nutzen, vom Eintreten dieses Ereignisses unberührt ist und daher weiterhin aktiv bleibt. Dies ist beispielsweise in UML generell der Fall: wie im letzten Absatz zitiert gehen Ereignisse verloren, wenn ihnen keine Transition zugeordnet ist.

[41] vgl. etwa [Pnu91]
[42] vgl. [HK99a], S. 130. Mit Transition ist hier selbstverständlich ein Übergang gemeint.
[43] vgl. [HK99a], S. 129, oder [Rum99], S. 73. Auch hier steht Transition für Übergang.

Dem steht die starke Rahmenbedingung entgegen, die fordert, dass jeder Zustand eine explizite Reaktion auf jedes Ereignis parat hält. Damit wird es notwendig, dafür zu sorgen, dass die Ereignisse, die auf einen Zustand keinen Einfluss haben sollen, nicht eintreten können. Beispielsweise könnte es sich um einen Knopf handeln, der das letzte Element einer Liste entfernt; wenn die Liste leer ist, besitzt er keinerlei Funktion mehr. Für diese Fälle sollte ein derartiges Dialogelement unsichtbar gemacht werden. In der Literatur wird hierfür meist eine separate Tabelle vorgeschlagen, in der zu jedem Dialogelement angegeben ist, in welchen Zuständen sie sichtbar (*enabled/visible*) beziehungsweise unsichtbar (*disabled/invisible*) sind.[44]

2.6 Bedingungen

Bedingungen werden in der Literatur allgemein als geschlossene boolesche Ausdrücke beschrieben.[45] Sie sind in diesem Sinne aus atomaren Bedingungen mittels der Operatoren AND, OR und NOT (sowie Klammern) zusammengesetzt und müssen erfüllt sein, um den zugehörigen Übergang zu ermöglichen. Über atomare Bedingungen wird nicht viel ausgesagt, allerdings treten sie meist als einfache Abfragen von Variablen auf, wie etwa `X>0`, oder als Tests, ob bestimmte Zustände des Statecharts gerade aktiv sind (beispielsweise `in?(Z)`).

Darüber hinaus sind auch Bedingungen denkbar, deren Wahrheitswerte nicht aus dem Statechart selbst abgeleitet werden können. Analog zu den Ereignissen kann man also von internen und externen Bedingungen sprechen. Für die theoretische Betrachtung der Statecharts ist diese Unterscheidung nebensächlich; beim Modellieren eines Systems müssen diese Bedingungen jedoch gesondert betrachtet werden.[46]

Meist wird vorausgesetzt, dass die Bedingungen, die den Austritt aus einem Zustand regeln, in Kombination mit dem auslösenden Ereignis dafür sorgen, dass nur jeweils ein Übergang möglich ist, und dadurch das System deterministisch machen. Dies ist insbesondere bei transienten Zuständen oder den Ereignisknoten in UML der Fall. Dabei ist meist die spezielle Bedingung `else` erlaubt, die nur dann wahr ist, wenn alle anderen Bedingungen nicht erfüllt sind.[47] Nichtdeterminismus kann darüber hinaus durch Prioritäten an den Übergängen ausgeschlossen werden; dies wurde bereits in Abschnitt 2.4 besprochen.

In einer der ersten Schriften über Statecharts schlägt Harel die Möglichkeit von Übergängen vor, die einer mehrwertigen Logik entsprechen.[48] Präziser wäre es, dies in die Sprache der Bedingungen einzubauen, da diese die Übergänge kontrollieren. Eine weitere naheliegende Verallgemeinerung der Bedingungen wäre die Verbindung mit temporaler Logik. Doch da in der Literatur einheitlich klassische Logik Verwendung findet, wird darauf nicht weiter eingegangen.

[44] vgl. beispielsweise [Hor99a], S. 46
[45] vgl. z. B. [HN96], S. 297
[46] vgl. Abschnitt 94
[47] vgl. [HK99a], S. 156
[48] vgl. [Har88], S. 526 f.

2.7 Aktionen

Aktionen werden im Statechart dadurch ausgelöst, dass ein Übergang stattgefunden hat. Wie schon bei Ereignissen und Bedingungen lassen sich interne Aktionen, die sich auf das Statechart selbst beziehen, von externen Aktionen unterscheiden, die das restliche System oder dessen Umwelt beeinflussen.

In der Literatur werden einige interne Aktionen genannt und verwendet. Dazu gehören insbesondere das Auslösen von weiteren (internen) Ereignissen und das Löschen der History eines Zustands.[49] Die ausgelösten Ereignisse werden im diskreten Zeitmodell im nächsten Schritt in gleicher Weise behandelt wie externe Ereignisse, das heißt sie können weitere Übergänge auslösen.

Es gibt zwei Arten von Aktionen, die Histories löschen. Die einfache Version bezieht sich lediglich auf einen einzelnen Zustand, die andere auch auf alle seine Unterzustände. Diese Formen erinnern zwar an die zwei Formen des Historyeinstiegs, sie sind von diesen jedoch unabhängig. Es ist durchaus erlaubt, die History eines Zustands zu löschen, der einen Deep-Historyeinstieg besitzt, ohne seine Unterzustände zu beeinflussen.

Harel und Namaad nennen die Möglichkeit, Aktionen zu verzögern (*scheduled actions*), also zu einem späteren Zeitpunkt auszuführen.[50] Für solche Aktionen wird angegeben, um wie viele Zeitschritte später sie ausgeführt werden sollen.

Eine denkbare Anwendung solcher verzögerter Aktionen wäre es, bestimmte Ereignisse nach einer gewissen Zeit auszulösen. Diese Konstruktion unterscheidet sich jedoch von den oben genannten aufgeschobenen Ereignissen in UML. Bei verzögerten Aktionen ist es nicht möglich, den „frühest möglichen“ Zeitpunkt als Termin anzugeben, wie es bei aufgeschobenen Ereignissen der Fall ist. Andererseits kann bei solchen Ereignissen keine minimale Zeitspanne vor der Ausführung angegeben werden.

2.8 Der Zeitbegriff

2.8.1 Kontinuierlich, synchron oder asynchron

Einer der grundlegendsten Punkte, in dem sich die beschriebenen Formen der Statecharts bei einzelnen Autoren unterscheiden, ist das Zeitmodell. In diesen Zusammenhang gehört einerseits die Frage, ob von einer Taktung des Systems ausgegangen wird, und damit eine diskrete Zeit vorliegt, oder ob die Zeit kontinuierlich verläuft. Bei getakteten Statecharts muss darüber hinaus entschieden werden, ob jeder Schritt des Statecharts tatsächlich einen Zeitschritt bedeutet, oder ob das Statechart mehrere Übergänge nacheinander ausführen kann, bevor die Uhr einen Zähler weiterspringt.

Wie bereits erwähnt, geht UML von einem kontinuierlichen Zeitbegriff aus. Ereignisse werden im Moment ihres Entstehens an die aktiven Zustände weitergegeben und eventuelle Übergänge ausgelöst; das Ereignis verschwindet daraufhin wieder. Dieses Verhalten wurde bereits in Abschnitt 2.5.1 genauer beschrieben. Um dennoch sicherzustellen, dass einige Teile des Statecharts aufeinander warten oder gleichzeitig

[49] vgl. z. B. [Har88], S. 523, oder [HN96], S. 306
[50] vgl. [HN96], S. 297 f.

ablaufen, wurde der Synch-Zustand eingeführt, der im Zusammenhang mit Konnektoren bereits in Abschnitt 32 besprochen wurde.

Die beiden erwähnten Formen von diskreter Zeit nennen Harel und Namaad synchron (höchstens ein Übergang des Statecharts pro Zeiteinheit) beziehungsweise asynchron (mehrere Übergänge des Statecharts sind pro Zeiteinheit möglich).[51] In [HN96] gehen sie allerdings nicht genauer auf die Unterschiede der beiden Modelle ein, sondern beschreiben hauptsächlich, welche Befehle der Simulator von Statemate zur Verfügung stellt; derartige Betrachtungen gehören jedoch nicht direkt zur Beschreibung von Statecharts und sollen deshalb auf Abschnitt 92 verschoben werden, in dem die Anforderungen an einen Simulator genauer diskutiert werden.

2.8.2 Delays und Time-Outs

Verzögerungen und Ereignisse, die nach einer bestimmten Zeitdauer eintreten, werden in der Literatur in vielen unterschiedlichen Fassungen vorgestellt. Dazu gehören verzögerte Aktionen (*scheduled actions*) und verzögerte Ereignisse (*deferred events*), die in den Abschnitten 2.7 beziehungsweise 2.5.1 besprochen wurden.

Darüber hinaus gibt es Ereignisse, die im Prinzip von einer Stoppuhr erzeugt werden, und selbst Übergänge auslösen können: die sogenannten *Time-Out*-Ereignisse. Harel benutzt diese bereits in seinen ersten Veröffentlichungen zu dem Thema im Beispiel einer Armbanduhr, die nach zwei Minuten ohne Tastendruck zur Zeitanzeige zurückschaltet.[52] In dem von Harel gezeigten Statechart wird dies einfach durch die Angabe `nach 2 Min` anstelle eines Ereignisse am Übergang deutlich gemacht. Die interne Stoppuhr wird dabei bei jedem Tastendruck wieder auf Null gesetzt,[53] und wenn die angegebene Zeit verstrichen ist, wird der entsprechende Übergang ausgeführt. In [HN96] geben Harel und Namaad eine genauere Definition der Funktionsweise eines Time-Out-Ereignisses:[54]

> „Das spezielle Ereignis `timeout(e,d)` tritt `d` Zeiteinheiten nach dem letzten Eintreten des Ereignisses `e` ein."

Interessant ist dabei zu bemerken, dass sich diese Ereignisse beim asynchronen Zeitmodell nicht auf Schritte im Statechart beziehen, sondern auf die simulierte Zeit. Es kann daher nicht vorher festgelegt werden, wie viele Übergänge im Statechart bis zum Eintreten des Ereignisses wirklich vergangen sind. Außerdem handelt es sich um reguläre Ereignisse, das heißt sie sind im gesamten Statechart sichtbar. Dies steht im Gegensatz zu der Form, die im Zusammenhang mit der Armbanduhr beschrieben wurde, da in jenem Fall kein Ereignis im eigentlichen Sinn benutzt wurde.

Horrocks schlägt ein etwas anderes Konzept vor, indem er Verzögerungen und Time-Outs in die Zustände integriert.[55] Eine Verzögerung bewirkt dabei, dass Eingaben des Benutzers nicht vor einer gewissen Zeit beachtet werden, da das System den Zustand nicht vor Ablauf der Frist verlassen kann. Die Zeitspanne eines Time-Out-Ereignisses wird ebenfalls an einem Zustand notiert, und jedes Mal, wenn dieser

[51] vgl. [HN96], S. 316 ff.
[52] vgl. [Har87b], S. 237, oder [Har88], S. 524 f.
[53] Diese Aktion ist im Statechart allerdings nicht dargestellt, sondern nur im erklärenden Text erwähnt.
[54] vgl. [HN96], S. 297 f.
[55] vgl. [Hor99a], S. 72 ff.

Zustand betreten wird, wird der Zähler auf Null gesetzt. Ist die angegebene Zeit verstrichen, wird der Zustand über eine Verbindung verlassen, die mit der Bezeichnung `time-out` versehen ist. Es wird dabei kein spezielles Ereignis erzeugt, wie es bei Harel und Namaad der Fall ist.

Ask not what your country can do for you;
ask what you can do for your country.
(John F. Kennedy)

Kapitel 3

Werkzeuge für Statecharts

Dieses Kapitel beschäftigt sich damit, welche Werkzeuge zur Unterstützung der Arbeit an Statecharts denkbar sind. Es sollen die Vorteile dieser Werkzeuge genauer erläutert und die Grundanforderungen dafür aufgezeigt werden. Außerdem sollen mögliche Erweiterungen und Kombinationen dieser Werkzeuge diskutiert werden, sowie die Frage, wie aufwendig die Umsetzung der einzelnen Werkzeuge ist.

Die ersten beiden Abschnitte dieses Kapitels beschäftigen sich im einzelnen mit Werkzeugen zum Erstellen beziehungsweise Überprüfen von Statecharts; im dritten Abschnitt werden Überlegungen dazu angestellt, welche Vorgänge sich automatisieren lassen.

3.1 Grafische Editoren

3.1.1 Der Statechart-Editor

Das wohl grundlegendste Werkzeug bei der Konstruktion von Statecharts ist ein grafischer Editor, der genau auf diese Konstruktionsaufgabe zugeschnitten ist. Selbstverständlich sollte ein solcher Editor die Grundfunktionen eines Editors besitzen, wie etwa Laden und Speichern, Hinzufügen, Modifizieren und Löschen der Elemente eines Statecharts und so weiter. Für die unterschiedlichen Editierfunktionen sollte es geeignete Knöpfe geben, mit denen der Benutzer das Statechart komfortabel bearbeiten kann. Da es bereits eine Vielzahl von guten grafischen Editoren gibt, soll hier auf den Aufbau einer leicht benutzbaren Oberfläche nicht weiter eingegangen werden. Wir wollen uns stattdessen einigen Details zuwenden, die in dieser Form lediglich bei Statechart-Editoren auftreten.

So ist etwa das Speichern und Laden von Statecharts aufwendiger als das von Text oder unabhängigen grafischen Objekten, da Statecharts eine nichtlineare Struktur besitzen. Es stellt sich von vornherein das Problem, wie die Zustände und ihre Zusammenhänge im Editor am besten verwaltet werden.

Die hierarchische Struktur, die für Statecharts typisch ist, wird am natürlichsten durch einen Baum beschrieben. Dabei steht allerdings bei keinem Knoten fest, wie viele direkte Nachfolger er haben kann. Bei atomaren Zuständen, die den Blättern des Baumes entsprechen, gibt es selbstverständlich keine weiteren Nachfolger; OR-

Zustände können hingegen beliebig viele Nachfolger besitzen. Bei AND-Zuständen stellt sich zusätzlich die Frage nach der Verwaltung der orthogonalen Komponenten. Es ist einerseits denkbar, die in den Komponenten enthaltenen Zustände als direkte Unterzustände des AND-Zustands anzusehen und zu verwalten; dann muss die Information, wie die Partitionierung stattfindet, separat gespeichert werden.

Andererseits hat bei genauerer Betrachtung eine einzelne Partition genau die Eigenschaften eines OR-Zustands: so muss etwa genau einer ihrer direkten Unterzustände aktiviert sein, wenn die Partition (also der AND-Zustand) aktiv ist, sie darf genau einen Defaulteinstieg besitzen etc. Es bietet sich also an, einen AND-Zustand so zu modellieren, dass jede Partition als Unterzustand gespeichert wird, und die eigentlich direkten Unterzustände des AND-Zustands als Unterzustände dieser Partitionen angesehen werden. Dies erfordert natürlich eine entsprechend aufwendigere Behandlung, wenn die Tiefe eines Zustands unter einem anderen festgestellt werden soll, da die Partitionen keine eigene Hierarchieebene darstellen dürfen.

Die Übersichtlichkeit der Darstellung

Durch den im letzten Abschnitt erwähnten Baum ist die logische Struktur des Statecharts bereits vollständig beschrieben. Die Lage der Unterzustände in der grafischen Ansicht ist dadurch allerdings noch nicht bestimmt. Das Problem der Positionierung stellt sich schon dadurch, dass die Unterzustände eines OR-Zustands nicht willkürlich verteilt sein sollten, und ein AND-Zustand grafisch kompliziert in Partitionen unterteilt werden darf.

Bei komplizierten Statecharts mit vielen Hierarchieebenen wird allerdings selbst bei guter Konstruktion der Überblick schnell verloren gehen, wenn der Benutzer ständig sämtliche Details angezeigt bekommt. Es sollte daher die Möglichkeit geben, eine Auswahl zu treffen, was angezeigt werden soll.

Denkbar ist zum Beispiel, dass nur bestimmte Typen von Elementen angezeigt werden, wie etwa Zustände oder Pfeile. Dies ist jedoch nur bei wenigen Typen sinnvoll, da bei ausgeblendeten Übergängen die Dynamik des Statecharts nicht mehr erkennbar ist; und kaum ein Benutzer wird ein Statechart verstehen können, ohne die Zustände zu sehen. Andererseits kann es durchaus hilfreich sein, die im Statechart notierten Texte zu verbergen, da dadurch die Struktur klarer werden kann.

Eine andere Möglichkeit ist es, die Ansicht auf gewisse Hierarchieebenen zu beschränken. Dabei werden nicht für alle Zustände auch die darin befindlichen Unterzustände angezeigt, sondern Zustände, die in der Hierarchie zu tief liegen, vernachlässigt. Diese Eigenschaft der Ansicht wollen wir hier mit dem Begriff *Tiefe* oder *Tiefenvergrößerung* bezeichnen. Es sollte darauf geachtet werden, dass die Auswahl der Pfeile, die angezeigt werden, dieser Tiefenvergrößerung angepasst sein sollte.

Angenommen, die Konstruktion entspricht der in Abbildung 3.1 (a) gezeigten, und nun wird die Darstellungstiefe derart verringert, dass die Zustände `C`, `D` und `E` unsichtbar werden. Es ist nun nicht sinnvoll, eine Darstellung wie in (b) zu erhalten, da beide Endpunkte der Verbindung t_1 unsichtbar sind und ein solcher Pfeil nur verwirrt. Andererseits ist es irreführend, die Verbindung t_2 ebenfalls auszublenden, wie es in (c) der Fall ist, obwohl auch ihre beiden Endpunkte nicht dargestellt werden, da nun die Zustände `A` und `B` völlig unverbunden scheinen. Die wünschenswerte Darstellung ist vielmehr die in (d) gezeigte.

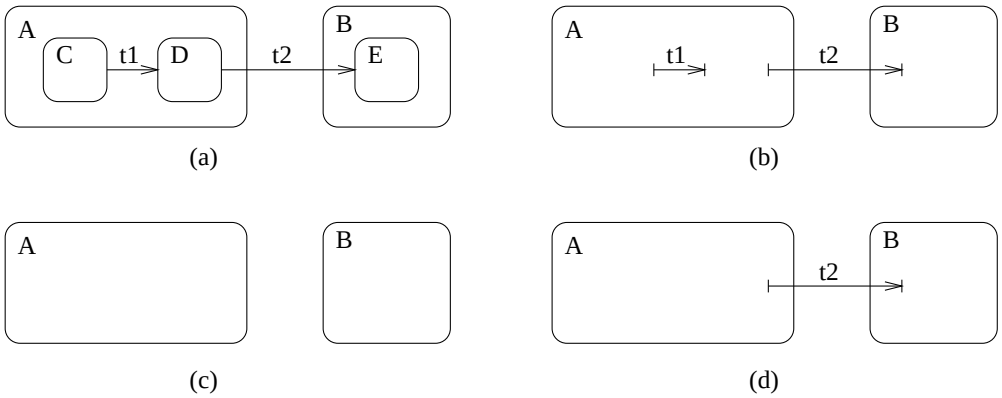


Abbildung 3.1: Darstellung von Pfeilen bei verschiedenen Tiefenvergrößerungen

Wichtig ist in der Darstellung auch, ob ein Pfeil *an* oder *in* einem Zustand endet beziehungsweise beginnt. Die in Abbildung 3.1 verwendete Notation wurde von Harel eingeführt,[1] um darzustellen, ob ein Zustand selbst oder einer seiner Unterzustände Teil der Verbindung ist. Während bei einer ausgehenden Verbindung der erste Fall darauf hindeutet, dass diese Verbindung unabhängig vom momentan aktiven Unterzustand des Startzustands ist, ist im zweiten Fall genau eine solche Abhängigkeit vorhanden. Analog wird bei eingehenden Verbindungen entweder der Defaulteinstieg verwendet oder ein explizit angegebener Unterzustand aktiviert.

Globale oder partielle Tiefenvergrößerung

Es kommt nun darauf an, ob der Benutzer die Möglichkeit geboten bekommt, lediglich einzelne Zustände in diese Tiefenvergrößerung einzubinden, oder das Statechart nur als ganzes in verschiedenen Hierarchieebenen anzuzeigen. Die erste Variante ist deutlich komplizierter, da die Größenverhältnisse der Zustände zueinander dabei nicht konstant sind. Wird nämlich die Betrachtungstiefe nur bei einem Zustand erhöht, braucht er eventuell mehr Platz als vorher, um die zusätzlichen Details aufnehmen zu können. Dadurch ändert sich natürlich die relative Lage der umliegenden Zustände und insbesondere die der Pfeile von oder zu diesem Zustand. Dies kann dazu führen, dass das Statechart gänzlich anders aussieht, und der Benutzer sich schwerer orientieren kann. Abbildung 3.2 verdeutlicht dies an einem Beispiel, in dem ein Statechart mit unterschiedlicher partieller Tiefe gezeigt wird. Die Konstruktion ist in beiden Bildern die selbe, doch diese Übereinstimmung kann nicht auf den ersten Blick festgestellt werden.

Obwohl es also zunächst selbstverständlich erscheinen mag, dass ein Editor die Möglichkeit einer solchen partiellen Tiefenvergrößerung bieten sollte, wird der Überblick dadurch deutlich erschwert: es wäre mit einem solchen Editor sehr kompliziert, die Zustände so anzuordnen, dass sie in allen Kombinationen der Tiefenvergrößerung ein übersichtliches Statechart ergeben. Um dem Benutzer diese Aufgabe zu erleichtern, könnte der Editor die Zustände automatisch platzieren. Dadurch werden allerdings die

[1] vgl. [Har88], S. 527

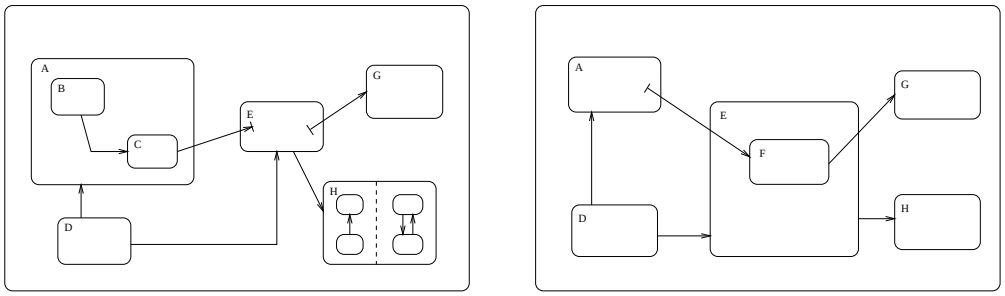


Abbildung 3.2: Partieller Tiefenvergößerung bei den Zuständen A und H bzw. E

Freiheiten des Benutzers in der grafischen Anordnung deutlich eingeschränkt; eventuell werden sogar eingegebene Positionen ignoriert, obwohl dies nicht die Absicht des Benutzers war. Abgesehen davon führt das auf ein Optimierungsproblem, das bekanntermaßen NP-hart ist.[2]

Die Variante, lediglich eine globale Tiefenvergrößerung zu erlauben, also die Anzahl der eingeblendeten Hierarchieebenen über das gesamte Statechart gleich zu halten, umgeht diese Probleme. Dabei muss jedoch noch die Frage geklärt werden, welche Größe atomare Zustände auf den verschiedenen Hierarchieebenen haben. Einerseits kann eine konstante Größe für atomare Zustände angenommen werden, die unabhängig von der Vergrößerung des Statecharts ist. Andererseits ist es denkbar, dass auch atomare Zustände größer angezeigt werden, wenn mehr Hierarchieebenen ausgewählt sind – diese Zustände blieben dann einfach weiterhin leer. Dies kann jedoch wieder den Überblick erschweren, da dabei große leere Flächen im Statechart auftauchen können. Es ist daher am sinnvollsten, den Benutzer die Größe der einzelnen Zustände selbst wählen zu lassen.

Es gibt allerdings noch eine weitere Möglichkeit, verschiedene Darstellungstiefen gleichzeitig anzuzeigen: es können mehrere Ansichten auf das Statechart in verschiedenen Fenstern zugelassen werden, die – soweit es die Tiefenvergrößerung betrifft – voneinander unabhängig sind. Wichtige Ausschnitte können dadurch vergrößert dargestellt werden, während der Rest entweder gar nicht oder übersichtlich in geringer Tiefe angezeigt wird.

Dieses Konzept entspricht darüberhinaus dem Vorgehen bei der Entwicklung eines Statecharts. Einzelne Zustände oder Partitionen stellen in vielen Fällen modulare Einheiten dar, die entweder keine oder nur wenige Verbindungen in andere Teile des Statecharts besitzen. Solange dieser Abschnitt des Statecharts konstruiert wird, ist es daher nicht nötig, das komplette Chart zu sehen. Hier ist eine Ansicht des momentan bearbeiteten Zustands sinnvoll, bei der eine große Tiefe eingestellt ist. Das fertige Modul kann dann in das globale Statechart eingefügt werden; hierbei reicht es meist aus, das Modul nur als einfachen Zustand ohne feinere Struktur zu sehen.

[2] vgl. [Sch93], S. 155: Das genannte Optimierungsproblem lässt sich mittels des entsprechenden Schwellwertproblems auf das *Bin-Packing-Problem* zurückführen.

Flexible Verbindungen

Ein weiteres Problem eines Statechart-Editors ist das Ziehen der Verbindungen. Wird partielle Tiefenvergrößerung erlaubt, kann sich die Lage zweier Zustände zueinander deutlich ändern, wenn diese in unterschiedlicher Darstellungstiefe angezeigt werden. Verbindungen zwischen diesen Zuständen müssen mit diesen Veränderungen derart umgehen können, dass das angezeigte Bild für den Benutzer immer sinnvoll ist. Es ist daher nicht ratsam, den Benutzer in diesem Fall Verbindungen durch frei gezeichnete Pfeile in das Statechart einfügen zu lassen. Andererseits ist es nicht einfach, Verbindungen durch die Lage der Zustände automatisch bestimmen zu lassen. So muss etwa darauf geachtet werden, dass Pfeile zwischen zwei Zuständen nicht plötzlich durch einen dritten hindurchgehen, oder andere Verbindungen derart schneiden, dass nicht mehr erkennbar ist, welcher Anfang zu welchem Ende gehört. Darüber hinaus sollten die Pfeile beschriftet sein, und diese Beschriftungen dürfen sich auch dann nicht überlagern, wenn beispielsweise zwei Zustände durch mehr als einen Pfeil miteinander verbunden sind.

Werden verschiedene Ansichten auf das Statechart angezeigt, die unterschiedliche Darstellungstiefen besitzen, so sind zwei Möglichkeiten denkbar, wie dies mit dem Ziehen von Pfeilen während der Konstruktion verbunden werden kann. Einerseits kann ein Pfeil in einer Ansicht begonnen und in einer anderen beendet werden. Dies hat den Vorteil, dass der Benutzer die Zustände direkt so verbinden kann, wie es letzten Endes gemeint ist. Allerdings ist die Verbindung von zwei Fenstern in einer Aktion nicht ganz übersichtlich, da es hierbei – im Unterschied etwa zu *Drag-and-Drop*-Aktionen zwischen zwei Fenstern – auch auf die Verbindungsstrecke zwischen Start- und Endpunkt der Aktion ankommt. Andererseits kann der Pfeil in einer gröberen Ansicht eingefügt werden, wo er lediglich an einen Zustand angehängt wird, der ein Oberzustand des eigentlichen Start- oder Endzustands ist. In einer detaillierteren Darstellung kann er dann so umgehängt werden, dass er den beabsichtigten Zustand anbindet.

Pfeile mit mehreren Start- oder Endpunkten

Weiterhin muss überlegt werden, wie Pfeile eingefügt werden können, die bei mehreren Zuständen beginnen oder enden. Dies sollte sich im Vorgehen nicht zu sehr vom Einfügen von Pfeilen unterscheiden, die einen einzigen Startzustand mit einem einzigen Endzustand verbinden. Wird für derartige „einfache“ Pfeile die Maus so benutzt, dass der Knopf auf dem Startzustand gedrückt und auf dem Endzustand losgelassen wird, so ist das folgende Verfahren denkbar: Zunächst werden alle Startzustände markiert, dann wird eine Verbindung gezogen, alle Endzustände markiert und schließlich dem Editor mitgeteilt, dass die Aktion abgeschlossen ist. Dieses Vorgehen hat jedoch – wie alle Methoden, die sämtliche Start- bzw. Endzustände lediglich als Zustandsmengen übergeben bekommen – den Nachteil, dass der grafische Verlauf der eingefügten Verbindung relativ willkürlich ist. Es lässt sich beispielsweise nicht angeben, wo die Abschnitte des Pfeils sich treffen, oder ob in einigen Bereichen Zustände umgangen werden müssen, die auf dem direkten Weg stören.

Eine bessere Methode ist es, Pfeile durch ausgewählte Punkte in kleinere Abschnitte einzuteilen. Dies löst auf der einen Seite das Problem der Positionierung, da derartige Punkte (die wir *Hooks* nennen wollen) beliebig verschoben werden können.

Der Übergang verbindet sie in der festgelegten Reihenfolge, und kann so vom Benutzer ganz beliebig durch das Statechart geführt werden. Mit diesen Hooks lassen sich nun sämtliche Pfeile durch die im letzten Abschnitt beschriebene Anklicken/Loslassen-Methode zeichnen. Wird etwa ein Übergang benötigt, der bei zwei Zuständen `A` und `B` beginnt und bei einem Zustand `C` endet, so wird zunächst ein Pfeil zwischen den Zuständen `A` und `C` gezeichnet, ein Hook auf diesem Pfeil eingefügt, und dann der Zustand `B` mit diesem Hook verbunden. Dieses Vorgehen wird in Abbildung 3.3 (a) bis (c) dargestellt. In einem vierten Schritt kann dann der Pfeil so positioniert werden, dass er den Wünschen des Benutzers entspricht.

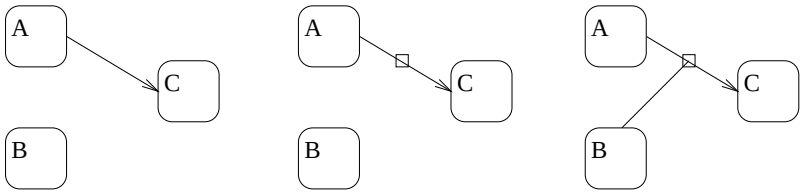


Abbildung 3.3: Einfügen eines 2-1 Pfeiles

Der Inspektor

Der Aufbau der Zustandshierarchie mit dem grafischen Editor und das Ziehen der Verbindungen zwischen Zuständen wurde in den letzten Abschnitten beschrieben. Das Statechart enthält aber noch viele weitere Informationen, die den dargestellten Objekten zugeordnet werden können. So können Zustände beispielsweise Ereignisse (oder Aktionen) auslösen, wenn sie betreten oder verlassen werden, möglicherweise dürfen sie auch statische Reaktionen enthalten, oder sie sind als Endzustand markiert. Sind darüberhinaus Parametrisierungen und Mehrfachinstantiierungen eines Zustands erlaubt, so müssen auch diese Informationen verwaltet werden. Ganz ähnlich können auch nicht alle Eigenschaften der Verbindungen im Statechart angezeigt werden, denn die meisten Bedingungen und Aktionen werden dafür zu lang sein. Es muss also eine Möglichkeit geben, diese Attribute, sowohl bei Zuständen als auch bei Verbindungen, außerhalb des Statecharts anzeigen und ändern zu können.

Besonders übersichtlich kann dies geschehen, wenn ein separates Fenster geöffnet wird, das Informationen zum momentan markierten Objekt des Statecharts darstellt. In einem solchen Fenster könnten diese Informationen dann auch geändert werden. Statt eines neuen Fensters ist diese Funktion (die wir im folgenden mit dem Begriff *Inspektor* bezeichnen wollen) auch durch einen reservierten Bereich des Hauptfensters möglich, sofern es ein solches gibt.

Die Informationen zu Zuständen und Pfeilen stellen selbstverständlich unterschiedliche Anforderungen. Während bei Zuständen etwa der Oberzustand interessiert, sollten bei Pfeilen Start- und Endzustände angegeben werden. Ein besonderer Komfort für den Benutzer wäre es, wenn er auch mittels des Inspektors durch das Statechart navigieren kann, das heißt, von einem markierten Pfeil aus dessen Startzustand markieren kann. Dies lässt sich recht naheliegend dadurch verwirklichen, dass derartige

Informationen im Inspektor auf Knöpfen dargestellt werden, durch die das entsprechende Objekt markiert werden kann.

Wenn der Editor die Möglichkeit bietet, nicht nur ein einzelnes Objekt markieren zu können, sondern auch mehrere, eventuell unterschiedliche Elemente des Statecharts, so muss der Inspektor darauf angemessen reagieren. Obwohl es denkbar ist, dass lediglich das erste markierte Objekt angezeigt wird, und die übrigen ignoriert werden, ist dies keine wünschenswerte Methode. Auf diese Weise würde der Inspektor nicht mehr der grafischen Ansicht entsprechen. Andererseits ist es kaum sinnvoll, alle markierten Objekte nebeneinander im Inspektor anzuzeigen, da hierdurch sehr viel Platz auf dem Bildschirm in Anspruch genommen werden kann. Vielmehr sollte der Inspektor anzeigen, dass das momentan detailliert dargestellte Statechart-Element nicht das einzige markierte Objekt ist. Dies ist beispielsweise durch die Benutzung der sogenannten Registerkarten möglich: jedes markierte Objekt wird durch eine Registerkarte repräsentiert, doch nur eine von diesen ist komplett lesbar.

Eine Überlegung zum Undo

Wie im letzten Abschnitt erwähnt, gibt es zu den einzelnen Elementen, aus denen ein Statechart aufgebaut ist, eine geraume Anzahl an Informationen. Es liegt daher nahe (insbesondere, wenn eine objektorientierte Programmiersprache verwandt wird), die Zustände und Übergänge als Objekte zu implementieren. Dies ermöglicht bei der *Undo/Redo*-Funktion eine interessante Variante.

Selbstverständlich sollte ein Statechart-Editor eine globale Liste der Änderungen an der Darstellung führen, so dass diese (oder wenigstens die letzte Änderung) zurückgenommen werden können; dies ist heutzutage bei den meisten grafischen Editoren möglich. Werden die Zustände jedoch als einzelne Objekte verwaltet, so ist es auch denkbar, dass jedes dieser Objekte eine eigene Liste von Änderungen verwaltet. So können Änderungen an einem Zustand zurückgenommen werden, während später vorgenommene Änderungen an einem anderen Zustand unberührt bleiben. Wird beispielsweise der Name eines Zustands geändert und darauf ein zweiter verschoben, wäre es nicht einsichtig, wenn die Änderung des Namens nicht zurückgenommen werden könnte, ohne dass auch die Verschiebung rückgängig gemacht wird.

Bekannte Verfahren Die Vorteile dieses Konzepts zeigen sich im Vergleich mit anderen bekannten Verfahren. Ein häufig implementiertes Konzept ist das, bei dem immer nur genau die letzte Änderung rückgängig gemacht werden kann; die Editiervorgänge werden sozusagen auf einen Stack abgelegt. Ein Redo wird dadurch ermöglicht, dass vom Stack genommene Vorgänge auf einem eigenen Stack gespeichert und dementsprechend in der richtigen Reihenfolge wieder hergestellt werden können. Neue Änderungen machen diesen Redo-Stack unzugänglich. Diese Form von Undo ist sehr übersichtlich, hat aber den eben beschriebenen Nachteil: vorhergehende Änderungen an Objekten, die in den letzten Vorgängen nicht bearbeitet wurden, können nicht rückgängig gemacht werden.

Dem gegenüber stehen Verfahren, bei denen sämtliche Vorgänge in einer Baumstruktur gespeichert werden, bei der ein neuer Ast begonnen wird, wenn nach einem Redo neue Aktionen durchgeführt werden. Hierbei gehen keine Editiervorgänge verloren; durch geschicktes Navigieren in diesem Baum ist es also möglich, frühere Ände-

rungen zurückzunehmen, und dann spätere Änderungen wiederherzustellen. Das Problem ist allerdings, dass der Benutzer nur schwer die Übersicht behält, an welcher Position im Baum er sich befindet.

Das hier erwähnte Konzept bietet die Übersichtlichkeit des ersten Verfahrens, kombiniert mit der Flexibilität eines baum-orientierten Undos. Es muss allerdings bei der Umsetzung noch genauer bestimmt werden, was bei Änderungen geschieht, die mehr als ein Objekt betreffen. Es ist leicht möglich, dass nach einer solchen Aktion vorangegangen Änderungen in mehreren der beteiligten Objekte nicht mehr zurückgenommen werden können.

Außerdem sollte auch das Löschen eines Zustands zurückgenommen werden können, so dass dieser Zustand durch das Löschen nicht unzugänglich werden darf. Dieses Problem ließe sich auf unterschiedliche Arten lösen, wie etwa durch die oben erwähnte zusätzliche globale Liste der Änderungen (oder auch nur Löschungen) oder einen Papierkorb, aus dem gelöschte Zustände wieder hervorgeholt werden können.

Ereignisse: aktiv oder passiv

Ereignisse werden in der Literatur einheitlich dafür verwendet, Übergänge auszulösen. Unter genauerer Betrachtung stellt sich heraus, dass dieses Auslösen auch mittels einer Bedingung modelliert werden kann, wie etwa `occurred?(event)`. Durch eine solche Bedingung könnte man diese Abfrage mit in den Bedingungsteil übernehmen. Damit ist es insbesondere möglich, die Ereignisse in komplexere Anfragen einzubinden und auch das Nicht-Eintreten eines Ereignisses zur Bedingung zu machen.

Diese Entscheidung hat jedoch weitreichende Konsequenzen auf den Quelltext des modellierten Systems.[3] Kann davon ausgegangen werden, dass für jeden Übergang zunächst ein Ereignis notwendig ist, das diesen Übergang auslöst, so kann sich das System den Großteil der Laufzeit in einer Art Wartezustand befinden, bis ein Ereignis gemeldet wird. Dies ermöglicht relativ schnelle Reaktionen des Systems. Werden die Ereignisse jedoch in die Bedingungen integriert, so müssen sämtliche ausgehenden Verbindungen aller aktiven Zustände kontinuierlich überprüft werden. Dies kann bei großen Systemen und komplexen Bedingungen durchaus die Reaktionsgeschwindigkeit beeinträchtigen.

Ein ähnliches Problem ergibt sich aus der Frage, ob auch verneinte Ereignisse als Auslöser zugelassen sind. Harel und Namaad schreiben dazu:[4] „Die Erfahrung zeigt, dass praktisch keiner der Benutzer [von Statemate] negierte Ereignisse benutzt hat, außer sie waren mit positiven Ereignissen durch UND verknüpft.“

Auf diese Feststellung ließe sich eingehen, indem das erwähnte „positive Ereignis“ als auslösendes Ereignis angegeben wird, und die Negation in den Bedingungsteil übernommen wird. Es wäre dann immer noch für jeden Übergang ein Ereignis notwendig, so dass die Software in der oben beschriebenen Weise effizient arbeiten könnte, und die Ereignisse dennoch deutlich flexibler benutzt werden können.

Als Mittelweg bietet es sich an, die Verbindungen, die von Ereignissen ausgelöst werden – sei es durch ein explizit genanntes Ereignis oder eines, das durch eine Bedin-

[3] Wir gehen in diesem Zusammenhang davon aus, dass das Statechart als Quelltext exportiert werden kann, wie es in Abschnitt 3.3.2 beschrieben wird.

[4] vgl. [HN96], S. 324

gung impliziert wird – gesondert zu kennzeichnen, und lediglich die restlichen kontinuierlich zu überprüfen.

Zwei weitere Stufen

Ein Editor in der bisher dargestellten Form ermöglicht bereits das vollständige Erstellen eines Statecharts, wie es in der formalen Definition beschrieben wird. Mit einem Simulator, wie er in Abschnitt 3.2.2 auf Seite 77 beschrieben wird, lässt sich die Funktionsweise des so entworfenen Statecharts testen und das Aktivieren und Deaktivieren von Zuständen nachvollziehen.

Um dem System, das von diesem Statechart modelliert wird, jedoch auch die Möglichkeit der Interaktion mit der Außenwelt zu geben, muss dieser reine „Kern“ des Statecharts erweitert werden. So sollten die verwendbaren Ereignisse, Bedingungen und Aktionen sich nicht auf das Statechart beschränken, wie es bisher der Fall ist. In der Literatur wird von solchen Möglichkeiten so selbstverständlich ausgegangen, dass diesbezüglich keine genauere Betrachtung angestellt wird.

Die spezielle Erweiterung um Dialoge Wenn durch das Statechart beliebige Hardware beschrieben wird, was ohne Änderungen an der Definition möglich ist, so muss die Interaktion mit dem Benutzer natürlich in jedem System individuell ermöglicht werden – dies ist auf viele Arten denkbar und lässt sich nicht problemlos generalisieren. Bei Softwaresystemen, die mit dem Benutzer über Bildschirm, Maus und Tastatur in Verbindung stehen, wird die Kommunikation allerdings meist mittels eines Benutzerdialogs stattfinden. Die Komponenten, aus denen ein solcher Dialog aufgebaut ist, können bei der Konstruktion des Statecharts bereits eingeplant werden. Auf diese Weise wird der Statechart-Kern um einige spezielle Ereignisse, Bedingungen und Aktionen ergänzt, die sich direkt aus der Verbindung mit diesen Komponenten ergeben. So wird etwa durch das Drücken eines Knopfes ein neues Ereignis ausgelöst, es kann abgefragt werden, ob ein bestimmtes Feld angekreuzt ist, Textfelder können durch das Anzeigen eines bestimmten Textes auf eine Aktion reagieren und so fort. Um diese neuen Elemente auf einfache Weise in das Statechart integrieren zu können, ist die Kombination des reinen Statechart-Editors mit einem Dialog-Editor naheliegend – darauf soll in Abschnitt 3.1.2 genauer eingegangen werden.

Die maximale Erweiterung durch Quelltext Obwohl ein Editor, der die zusätzliche Benutzung von Dialogkomponenten bereits anbietet, deutlich mächtiger ist als ein reiner Statechart-Editor, lassen sich viele Probleme immer noch nicht komplett modellieren. Es fehlt eine Möglichkeit, mit anderen Komponenten oder Programmen des Systems zu kommunizieren. Hier gibt es wieder beliebig viele Varianten, wie zum Beispiel der Zugriff auf eine Datenbank, das Ansteuern von Hardware wie etwa CD-Spielern, die Kommunikation über ein Netzwerk und so weiter. Diese Funktionen sind so vielfältig, dass sie nicht zu weiteren speziellen Ereignissen und Aktionen zusammengefasst werden können. Sollen sie dennoch in das Statechart integriert werden, gibt es nur eine Lösung: der Benutzer muss die Möglichkeit erhalten, auch beliebigen Quelltext in das Statechart eingeben zu können. Hierfür bietet es sich an, eine Gruppe von Aktionen, Bedingungen und eventuell Ereignissen einzuführen, deren Funktionen vom Benutzer frei programmiert werden können. Der Code einer Aktion wird dann

ausgeführt, wenn diese Aktion aufgerufen wird, derjenige einer Bedingung wird benutzt, um den Wahrheitswert der Bedingung festzustellen, und durch so programmierte Ereignisse können dem System Änderungen an überwachten Eigenschaften oder Komponenten melden.

Wird Quelltext im Statechart erlaubt, so stellt sich jedoch die Frage, wo er benutzt wird. Bietet das System die Möglichkeit, das konstruierte Statechart zu simulieren, so könnte der Quelltext bereits dabei benutzt werden. Kann aus dem Editor heraus ausführbarer Programmtext erzeugt werden, der dem Statechart entspricht, so sollte insbesondere hierbei explizit eingegebener Quelltext beachtet werden. Die Möglichkeiten und Probleme, die sich daraus ergeben, sollen im Zusammenhang mit den entsprechenden Werkzeugen in Abschnitt 94 für die Simulation beziehungsweise Abschnitt 3.3.2 für den Quelltextexport genauer betrachtet werden.

In den folgenden Abschnitten wollen wir nun weitere Werkzeuge diskutieren. Dabei werden wir immer davon ausgehen, dass diese mit einem Statechart-Editor zusammenarbeiten, wie er hier beschrieben wurde.

3.1.2 Der Dialog-Editor

Statecharts modellieren interaktive Systeme. Obwohl ein solches System völlig ohne die Hilfe eines Computers auskommen kann, wird es sich doch oft um Software handeln, die mit dem Benutzer über Bildschirm, Tastatur und Maus in Verbindung steht. In diesem Fall geschieht die Kommunikation meist mittels eines Benutzerdialogs. Zum Entwurf eines solchen Dialogs ist es naheliegend, einen eigenen Editor anzubieten. Die Frage, wie ein Dialog-Editor das Erstellen eines allgemeinen Dialogs unterstützen kann, und das Problem, gute Benutzerschnittstellen zu erstellen, sollen hier jedoch nicht besprochen werden. Vielmehr geht es darum, welche Funktionen ein Dialog-Editor im Zusammenhang mit Statecharts bieten sollte.

Der Einfachheit halber sei hier angenommen, dass es sich lediglich um einen einzigen Benutzerdialog handelt. Die Überlegungen lassen sich jedoch direkt auf mehrere Dialoge erweitern.

Interaktion

Im allgemeinen wird der Dialog mit dem Statechart über Ereignisse, Bedingungen und Aktionen in Verbindung stehen: Wird ein Knopf des Dialogs gedrückt oder ein Feld ausgewählt, so sendet der Dialog ein Ereignis an das System, das mittels des Statecharts modelliert wird. Das Statechart (oder vielmehr das System) reagiert darauf mit einem Übergang in einen anderen Zustand und löst gegebenenfalls Aktionen aus. Diese Aktionen können unter anderem dazu führen, dass sich der Dialog ändert, indem etwa weitere Knöpfe sichtbar oder Textfelder gelöscht werden. Über Bedingungen können auch Anfragen an den Dialog formuliert werden, wie etwa die Frage, ob ein bestimmtes Textfeld momentan leer ist.

Der Benutzer braucht eine komfortable Möglichkeit, diese Zusammenhänge zu formulieren und nachzuvollziehen. Der Dialog-Editor muss also deutlich machen, welche Ereignisse von den einzelnen Komponenten des Dialogs erzeugt werden, und diese sollten vom Benutzer auf einfache Weise in das Statechart übernommen wer-

den können. Ähnlich zum Inspektor, der im Abschnitt 3.1.1 für die Elemente des Statecharts beschrieben wurde, ließe sich hier ein Inspektor für die Dialogkomponenten vorstellen. Darin sollten die verschiedenen Ereignisse angegeben sein, die von der ausgewählten Komponente erzeugt werden können, und die Aktionen, durch die sie beeinflusst werden kann. Wenn der Statechart-Editor dann ebenfalls einen Inspektor besitzt, könnte man hier etwa eine *Drag-and-Drop*-Funktionalität anbieten, durch die der Benutzer sich Ereignisse, Bedingungen und Aktionen im Statechart zusammenbauen kann.

Repräsentation

Das Statechart stellt den Zustand des Systems und daher auch den Zustand des Dialogs dar. In anderen Worten: den Komponenten des Dialogs, die ihren Zustand ändern können, entsprechen bestimmte Bereiche des Statecharts. So wird eine Checkbox[5] durch ein Paar von Zuständen (etwa `ON` und `OFF`) modelliert, wobei ein Klick auf die Checkbox einen Übergang zwischen diesen Zuständen bewirkt. Eine Gruppe von Radiobuttons[6] entspricht im Statechart einer Konstruktion wie in Abbildung 3.4.[7] Analog lassen sich Gruppen von Zuständen auch für andere Dialogkomponenten finden.

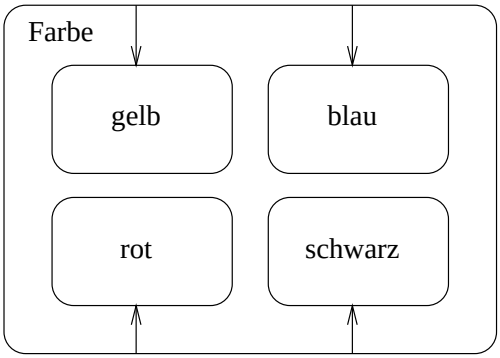


Abbildung 3.4: Radiobuttons im Statechart

Es ist wünschenswert, dass der Dialog-Editor beim Einfügen solcher Komponenten das Statechart entsprechend ergänzt oder zumindest beim Benutzer anfragt, ob und wo das Statechart um die neuen Zustände erweitert werden soll. Der Benutzer braucht dadurch diese Komponenten nur einmal und nicht in beiden Editoren zu ergänzen. Gleichzeitig ist sichergestellt, dass sich Dialog und Statechart entsprechen, und dass auf die vom Dialog erzeugten Ereignisse entsprechend reagiert wird.

Wenn die Logik des Dialogs auf diese Weise im Statechart kopiert wird, so lassen sich Anfragen nach der momentanen Auswahl des Benutzers innerhalb des Statecharts beantworten. Dadurch, dass die Modellierung einer Gruppe von Dialogkomponenten allerdings von vornherein festgelegt ist, kann es zur Übersichtlichkeit des Statecharts beitragen, wenn diese nicht explizit dargestellt wird. Es ist daher auch denkbar, dass

[5] Ein Feld, das entweder ausgewählt oder nicht ausgewählt ist

[6] Knöpfe, von denen immer genau einer gedrückt ist – beim Drücken eines nicht aktivierten Knopfes wird der bisher aktivierte deaktiviert

[7] Die Ereignisse an den Verbindungen wurden dabei nicht dargestellt.

der Dialog diese Information nicht in das Statechart überträgt, sondern selbst verwaltet und auf entsprechende Anfragen antwortet. Diese Variante widerspricht jedoch in einem gewissen Maß den Programmmodellen, die eine strikte Trennung von Benutzerschnittstelle und Datenverwaltung fordern. Darüberhinaus kann das Statechart dann nicht mehr getrennt vom Dialog betrachtet werden, da die Kommunikation zwischen ihnen nicht nur aus Ereignissen besteht, die vom Dialog ausgelöst werden.

Zugriff auf Dialogkomponenten

Aus der Erfahrung mit praktisch jedem Programm ist klar, dass nicht zu jedem Zeitpunkt alle Eingaben möglich sind. In guten Benutzerschnittstellen wird dies dem Benutzer dadurch angezeigt, dass die Knöpfe und Felder, die im Moment nicht akzeptiert werden, deaktiviert beziehungsweise unsichtbar sind. In der Sprache der Statecharts heißt dies beispielsweise, dass jedem Zustand ein Vektor zugeordnet werden kann, der angibt, welche Dialogkomponenten sichtbar sind, wenn der Zustand aktiviert ist, und welche nicht. Die übersichtlichste Weise, diese Vektoren darzustellen, ist vermutlich, sie in einer Tabelle zusammenzufassen. Wenn sowohl ein Statechart-Editor als auch ein Dialog-Editor vorliegen und daher klar ist, welche Zustände und welche Dialogkomponenten vorhanden sind, so sollte es auch eine Möglichkeit geben, diese Tabelle zu bearbeiten.

Eine andere Variante, Dialogkomponenten sichtbar und unsichtbar zu machen, ist es, diesen Zugriff durch Aktionen anzubieten. Obwohl in diesem Fall nicht aus dem aktivierten Zustand ablesbar ist, welche Dialogkomponenten sichtbar sind, fügt sich dieses Konzept leichter in die Struktur eines Statecharts ein. So muss keine zusätzliche Tabelle gespeichert werden, die Zustände und Dialog miteinander verbindet, sondern die Interaktion zwischen Statechart und Dialog kann weiterhin ausschließlich über Ereignisse und Aktionen stattfinden.

Kommunikation zwischen Statechart und Dialog

In diesem Zusammenhang stellt sich die Frage, wie das Statechart genau mit dem Dialog kommunizieren soll. Selbstverständlich ist es denkbar, für jede Dialogkomponente eigene Aktionen, Ereignisse und Bedingungen einzuführen. Mit Blick auf eine leichte Erweiterbarkeit des Programms um neue Klassen von Komponenten wäre jedoch eine vereinheitlichte Schnittstelle wünschenswert. Für jede Methode dieser Schnittstelle gibt es dann einen Typ von Aktion, der diese Methode bei einer bestimmten Dialogkomponente aufruft. Wie die Komponente auf einen solchen Aufruf reagiert (oder ob sie ihn eventuell sogar ignoriert), muss für jede Klasse von Komponenten einzeln spezifiziert werden. Es ist daher wünschenswert, die Zahl der Methoden möglichst gering zu halten.

Es sei zunächst erwähnt, dass die Kommunikation des Dialogs mit dem Statechart über Ereignisse geschehen sollte. Da die meisten Komponenten lediglich ein einziges Ereignis auslösen können (wie etwa „Knopf gedrückt“), sollte dies für alle Komponenten vorausgesetzt werden, um sie im Editor einheitlich behandeln zu können.[8] Bedingungen, deren Wahrheitswert aus dem Dialog abgelesen werden müssen, sind nicht notwendig, da sie im Statechart nachmodelliert werden können – dies wurde bereits im

[8] Auf Komponenten, die mehrere Ereignisse auslösen können, wird weiter unten eingegangen.

letzten Abschnitt besprochen. Wir wollen uns nun den Aktionen zuwenden, mit denen das Statechart auf den Dialog Einfluss nehmen kann.

Wie am Anfang dieses Abschnitts erwähnt wurde, sollten Dialogkomponenten sichtbar und unsichtbar gemacht werden können; für diese Funktionalität werden also zwei Methoden[9] benötigt. Werden im Dialog auch Popups[10] angeboten, so bietet sich diese Methode an, um das Fenster zu öffnen. Dies erlaubt es auch, das Popup mittels einer Aktion zu schließen, unabhängig davon, ob der Benutzer einen Knopf des Popups gedrückt hat.

Für Dialogkomponenten wie Checkboxes oder Radiobuttons wäre eine Methode wünschenswert, die einstellt, ob das Feld angewählt ist oder nicht. Dies ist besondern für Radiobuttons notwendig, wenn beim Drücken eines Knopfes das Deaktivieren der restlichen Knöpfe vom Statechart aus gesteuert werden soll.[11]

Variablen Besondere Schwierigkeiten treten im Zusammenhang mit Dialogkomponenten auf, die eine Zahl oder einen Text darstellen, wie etwa Schieberegler oder Textfelder. Hier sollte einerseits der Wert gesetzt werden können, andererseits sind viele Fälle denkbar, in denen unterschiedlichste Vergleiche mit diesem Wert durchgeführt werden müssen; er sollte also sowohl für den Dialog als auch für das Statechart zugänglich sein. Dieses Problem ist recht einfach zu lösen, wenn jeder Dialogkomponente eine Variable zugeordnet wird. Es muss allerdings darauf geachtet werden, dass der Wert der Variablen immer dem der zugehörigen Komponente entspricht. Mit diesem Verfahren lässt sich dann beispielsweise auch der Text eines Knopfes setzen – und sogar zur Laufzeit ändern und damit eventuell einer neuen Situation anpassen. Auch Fehlermeldungen können so sehr gut in das Statechart eingebaut werden: es muss lediglich ein einziges Popup zum Dialog hinzugefügt werden, das beim Auftreten eines Fehlers angezeigt wird. Die entsprechende Meldung kann in die zugeordnete Variable geschrieben werden.

Insbesondere lassen sich mit einer solchen Variable aber auch Dialogkomponenten modellieren, die eigentlich mehrere Ereignisse erzeugen können, wie etwa eine Ja/Nein-Anfrage. Hierbei lässt sich ein einziges Ereignis definieren („Auswahl getroffen“), und die Angabe, um welche Eingabe es sich handelt, wird dem Statechart in der Variablen der Dialogkomponente übergeben.

Obwohl natürlich noch beliebig viele weitere Methoden denkbar sind, dürften die meisten Dialogkomponenten durch eine derartige Schnittstelle zum Statechart in ausreichendem Umfang benutzbar sein.

3.1.3 Der Programmstruktur-Editor

Große Systeme, die ohne zu großen Aufwand gepflegt werden sollen, können praktisch nicht monolithisch sein; sie müssen aus Modulen und Untersystemen zusammengesetzt werden. Diese Struktur kann durch einen speziellen Editor parallel zu den Statecharts, die ihr Verhalten beschreiben, aufgebaut werden.

Ist nur ein Statechart vorhanden, so kann diese Struktur daraus nicht abgelesen werden. Es ist wohl nie sinnvoll, jeden Zustand als eigenständiges Objekt anzusehen,

[9] oder eine Methode mit zusätzlichem Parameter vom Typ *boolean*

[10] separate Fenster, die geöffnet werden, um eine Meldung anzuzeigen

[11] vgl. die Beschreibung der Radiobuttons im letzten Abschnitt

aber eine andere Gliederung des Statecharts ist zunächst nicht möglich. Um diese Unterteilung zusätzlich modellieren zu können, ist der Einsatz eines Programmstruktur-Editors empfehlenswert. Indem den Statecharts noch eine Ebene übergeordnet wird, können eng miteinander verbundene Teile zu eigenständigen Modulen zusammengefasst werden.

Dieses Modell ermöglicht es, das Verhalten eines Systems, bei dem einzelne Teile mehrfach auftreten können, komfortabel zu modellieren. Dies kann beispielsweise bei einem Statechart für eine Datenbank auftreten, zu der mehrere Benutzer über identische Schnittstellen gleichzeitig Zugriff haben sollen. Im Statechart sollte diese Schnittstelle nur ein einziges Mal modelliert werden, im ausführbaren Quelltext muss dieser Ausschnitt des Statecharts aber beliebig häufig nebeneinander existieren können. In anderen Worten muss dieser Ausschnitt mehrfach instanziiert werden können. Harel nennt dieses Verfahren AND-Parametrisierung.[12]

Beschränkt man sich darauf, lediglich einen Statechart-Editor zu verwenden, ist diese Aufgabenstellung mit einigen Problemen verbunden. Es stellen sich beispielsweise die Fragen, wie weit Ereignisse sichtbar sind, die in einem der mehrfach instanziierten Teile auftreten, und wie die Ereignisse einer Instanz von denen anderer Instanzen unterschieden werden können. Zunächst ist auch nicht offensichtlich, auf welche Instanz sich etwa eine Bedingung in einem übergeordneten Zustand beziehen soll. Außerdem müssen einzelne Instanzen beendet werden können, ohne dass dies zu Unstimmigkeiten im Programmablauf führt.

Werden auch Module des Programms modelliert, stellen sich diese Fragen nicht, da jedes Modul im Prinzip ein eigenständiges System ist, und die zugehörigen Statecharts voneinander unabhängig sind. Soll ein Teil des Systems mehrfach instanziiert werden, so kann man dieses Modul aus dem Rest des Systems einfach auskoppeln.

Hierbei ist natürlich gesondert festzulegen, wie die einzelnen Module miteinander kommunizieren. Es ist einerseits denkbar, beispielsweise die Ereignisse, die von einem Modul an ein bestimmtes anderes weitergegeben werden sollen, über eine Eingabe-Methode des empfangenden Moduls zu übertragen. Das erfordert jedoch, dass jedes einzelne Modul Zugriff auf alle anderen Module hat, mit denen es kommunizieren soll. Andererseits wäre eine Art Bus-System denkbar, in das Ereignisse eingespeist und von den Modulen, die darauf reagieren sollen, abgeholt werden. Welche dieser Varianten implementiert wird, steht in engem Zusammenhang mit der Rahmenbedingung, die für das Statechart gelten soll.[13]

Wichtig ist jedoch auch die Frage, wie die einzelnen Module im Statechart-Editor getrennt werden. Wird eine sehr strikte Trennung vorausgesetzt, so wäre es am sinnvollsten, die einzelnen Module nicht im selben Statechart anzuzeigen, sondern sie wie einzelne Statecharts zu behandeln. Ist die Trennung weniger streng, so dass Pfeile etwa Modulgrenzen kreuzen können, so entstehen zusätzliche Probleme. Es wird beispielsweise ein globaler Namensraum notwendig, um die Bezüge von Ereignissen und Aktionen innerhalb des Statecharts konsistent zu halten.[14]

[12] vgl. Abschnitt 2.2.3

[13] vgl. Abschnitt 2.5.2

[14] Für eine genauere Beschreibung dieser Schwierigkeiten sei auf den Abschnitt der Parametrisierung auf Seite 94 verwiesen.

3.2 Tester

3.2.1 Der Verifikator

Die grafische Sprache der Statecharts ist sehr flexibel. Doch nicht alle möglichen Bilder, die den Formalismen entsprechen, können auch als gültige Statecharts interpretiert werden.

Auf einige Fehler, die den Grundregeln widersprechen, kann der Statechart-Editor schon selbst hinweisen, beziehungsweise derartige Konstruktionen gar nicht erst ermöglichen. So sollten alle Übergänge ordnungsgemäß mit Zuständen verbunden sein, Konnektoren dürfen keine Unterzustände enthalten, etc. Die meisten möglichen Fehlerquellen können jedoch nicht auf so einfache Weise ausgeschlossen werden. Einige Konsistenzbedingungen können zum Beispiel erst nach einigen Editierschritten erfüllt werden, und daher kann ihre endgültige Nichterfüllung erst dann festgestellt werden, wenn das Statechart fertig definiert ist. Um diese Überprüfung vorzunehmen, ist ein weiteres Werkzeug notwendig, welches das konstruierte Statechart testet: der Verifikator. In den folgenden Abschnitten werden wir auf einige der interessanteren Probleme in diesem Zusammenhang eingehen.

Konstruktionsfehler

Es sind fast beliebig viele Testfragen denkbar, die durch ein solches Werkzeug überprüft werden sollten. Zunächst einmal sollten offensichtliche Konstruktionsfehler gemeldet werden. Um dem Benutzer das Konstruieren des Statecharts jedoch möglichst komfortabel zu machen, muss genau darauf geachtet werden, ob diese Fehler möglicherweise bereits vom Editor abgefangen und verhindert werden sollten. Einige Eingaben können grundsätzlich verboten werden, andere Unstimmigkeiten in der Konstruktion sind vorübergehend sogar unvermeidlich; hier kann erst nach Fertigstellung des Statecharts entschieden werden, ob immer noch Fehler vorhanden sind.

Wie bereits in Abschnitt 1.3.2 erwähnt, dürfen Verbindungen zum Beispiel keine Partitionsgrenzen schneiden. Werden Pfeile im Editor jedoch so eingefügt, dass sie Start- und Endpunkt zunächst in gerader Linie verbinden, so kann dies durchaus auftreten. Bei einem Editor, der mit den in Abschnitt 67 auf Seite 61 beschriebenen Hooks arbeitet, kann der Benutzer diesen Pfeil dann so modifizieren, dass er den Richtlinien entspricht (dieser Vorgang wird in Abbildung 3.5 dargestellt). Dies muss jedoch erst zum Ende der Konstruktion geschehen sein, so dass der Editor das Einfügen eines solchen Pfeils, der zunächst ungültig ist, nicht automatisch verhindern darf.

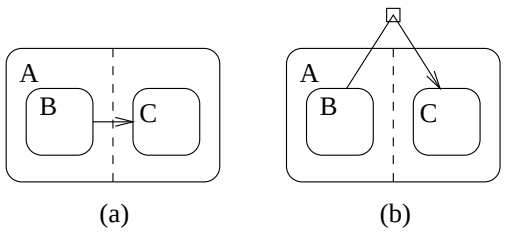


Abbildung 3.5: Ein zunächst ungültiger Pfeil wird korrigiert

Andere Probleme sind während der Konstruktionsphase nicht leicht zu überblicken; auch aus diesem Grund kann es unpraktisch sein, wenn der Editor einige Vorgänge nicht zulässt. Dazu gehören etwa Übergänge, deren Startzustände (oder Endzustände) in der Hierarchie übereinander liegen. Der Zugriff auf den oberen dieser Zustände ist in diesem Fall nicht sinnvoll, da die Verbindung auch ohne ihn dieselbe Reaktion des Statecharts auslösen würde. Besitzt die Zustandshierarchie allerdings viele Ebenen und sind vielleicht sogar Konnektoren in den Übergang eingebaut, so ist dieses Problem nicht mit einem Blick zu erfassen. Hier wäre eine ausführlichere Fehlermeldung hilfreich, die ein Verifikator leichter und passender als ein Editor anbieten kann.

Eine weitere denkbare Konstruktion, die zu Probleme führen kann, sind Gruppen von Konnektoren, die durch Übergänge zu einem Rundlauf zusammengeschlossen sind.[15] Wenn das Statechart einen solchen Kreis betritt und nicht verlassen kann, so kann es (in der Simulation des Statecharts oder auch einem deratig implementierten System) passieren, dass das System keinen gültigen Zustand erreichen kann und quasi gefangen ist. Es kann sein, dass die Übergänge eines solchen Kreise mit entsprechenden Ereignissen oder Bedingungen versehen sind, so dass einerseits das komplette Durchlaufen der Schleife nicht möglich ist und andererseits in jedem Fall ein Ausgang gefunden werden kann. Die Prüfung, ob die Pfeile diese Eigenschaft besitzen, ist jedoch nicht einfach, wie im nächsten Abschnitt beschrieben wird.

Abgesehen von der grafischen Konstruktion kann der Benutzer, wenn er beispielsweise für Aktionen beliebige Zeichenketten eingeben kann, auch hierbei fehlerhafte Eingaben machen. Um dies zu vermeiden, sollte entweder bereits der Editor, spätestens jedoch der Verifikator auch die Semantik von Ereignissen, Bedingungen und Aktionen verstehen.[16]

Determinismus

Problematisch kann bei einigen umfangreicheren Testfragen deren Komplexität werden. Man betrachte etwa einen Zustand mit mehreren ausgehenden Verbindungen, an denen unterschiedliche Bedingungen notiert sind. Das Problem, ob bei jeder denkbaren Belegung der in den Bedingungen auftretenden aussagenlogischen Variablen genau ein Übergang möglich ist (und das Statechart damit deterministisch ist), ist NP-vollständig.[17] In diesen Zusammenhang gehört auch das Problem einer Konnektorschleife, das im letzten Abschnitt angesprochen wurde. Die eigentlich interessante Fragestellung ist jedoch noch komplizierter: meist wird nicht jede denkbare Belegung der Variablen auch während der Abarbeitung des Statecharts auftreten können. Das Chart braucht jedoch lediglich für in diesem Sinne mögliche Belegungen deterministisch zu sein. Ebenso kompliziert ist die Frage, ob Bedingungen benutzt werden, die immer, sprich in jedem möglichen Fall, wahr oder falsch sind.

[15] Diese spezielle Konstruktion sollte ohnehin vermieden werden, da sie genausogut mit einem einzigen Konnektor modelliert werden kann, wenn sie überhaupt notwendig sein sollte.

[16] Dies wird auch schon in [Schu99] gefordert; vgl. beispielsweise dort S. 182.

[17] Das genannte Problem lässt sich auf das Erfüllbarkeitsproblem der Aussagenlogik (SAT) zurückführen, das nach dem Satz von Cook NP-vollständig ist; vgl. zum Beispiel [Sch93], S. 149. Es sei hier noch erwähnt, dass auch das UML-Werkzeug Rose der Rational Software Cooperation diese Fragestellung nicht überprüft, vgl. [Schu99], S. 185.

Glücklicherweise ist es einfacher zu überprüfen, ob von einem Zustand mehrere Verbindungen ausgehen, die von demselben Ereignis ausgelöst werden. Ist dies der Fall, kann zumindest eine Warnung an den Benutzer weitergegeben werden. Wenn dieser dann feststellt, dass sich die angegebenen Bedingungen gegenseitig ausschließen und das System deshalb durchaus deterministisch ist, kann er die Warnung beiseite legen.

Dass diese Frage bei größeren Statecharts allerdings weniger offensichtlich sein kann, als es zunächst klingt, soll das folgende Beispiel darstellen. So kann etwa in Abbildung 3.6 der Zustand `C` niemals erreicht werden, weil bei Eintreten des Ereignisses `e` der Zustand `A` bereits verlassen wird.[18]

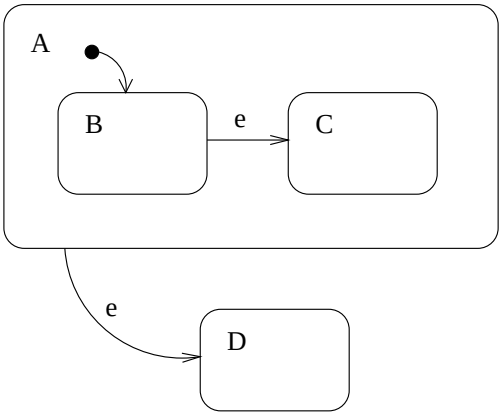


Abbildung 3.6: Konflikte bei identischen Ereignissen

Dieses Beispiel führt direkt auf die umfassendere Frage, ob alle Zustände des Statecharts erreichbar sind. Zu ihrer Beantwortung muss jede Folge von möglichen Zuständen durchgegangen werden, da sie sonst nicht verneint werden kann. Offensichtlich ist das Problem ähnlich umfangreich wie die bereits erwähnten. Der Verifikator kann jedoch wieder eine vereinfachte Variante überprüfen, indem er sicherstellt, dass jeder Zustand eingehende Verbindungen besitzt[19] oder zumindest als Defaulteinstieg seines Oberzustands markiert ist. Obwohl dies noch keine Garantie dafür ist, dass er wirklich erreichbar ist, stellt es doch eine notwendige Bedingung dafür dar.

Ebenso lässt sich überprüfen, ob jeder Zustand verlassen werden kann. Dafür reicht es allerdings nicht aus, dass jeder (nicht-terminale) Zustand ausgehende Verbindungen besitzt. Es könnte durchaus sein, dass das Ereignis, das diesen Übergang auslöst, zu einem Knopf gehört, der momentan unsichtbar ist. Damit kann das Ereignis nicht eintreten, und diese Verbindung ist praktisch nicht vorhanden. Allerdings kommt es hierbei darauf an, ob Dialogkomponenten anhand einer Tabelle oder mittels Aktionen sichtbar und unsichtbar gemacht werden.[20] In der tabellarischen Form lässt sich diese Bedingung relativ leicht abschließend überprüfen; in der zweiten Variante kann es durchaus sein, dass die notwendige Dialogkomponente zunächst unsichtbar

[18] Hierbei wird von einer Strukturellen Priorität ausgegangen, wie sie von Harel und Namaad für neuere Versionen von Statemate beschrieben wird; vgl. auch Seite 50.

[19] hierbei sind auch Verbindungen zu beachten, die auf Unterzuständen enden

[20] vgl. Seite 68

ist, aber später durch eine Aktion sichtbar gemacht wird. Offensichtlich kann dies nicht sichergestellt werden, ohne dass sämtliche Möglichkeiten durchgegangen werden.

Ein ähnliches Problem kann selbstverständlich auch auftreten, wenn das System in einer Gruppe von Zuständen gefangen wird. In diesem Fall ist jeder einzelne Zustand leicht zu verlassen, aber Zustände, die außerhalb dieser Gruppe liegen, können nicht wieder betreten werden. Obwohl dies zunächst nach einem Fehler klingt, ist es auch denkbar, dass genau dieses Verhalten gewünscht ist. Der Verifikator sollte also eine derartige Konstruktion zulassen, wenn sie vom Benutzer nicht als fehlerhaft angesehen wird.

Beim Test, ob Zustände verlassen werden können, muss darüberhinaus beachtet werden, dass ein Zustand auch dann verlassen wird, wenn dies bei einem Oberzustand geschieht. Bei AND-Zuständen reicht es sogar, dass ein von einer einzigen Partition ausgehender Übergang möglich ist. In Abbildung 3.7 kann beispielsweise jeder Zustand verlassen werden, obwohl dies bei den Zustände `D` und `E` nicht gleich offensichtlich ist. Zustand `D` wird etwa bei Eintreten des Ereignisses *f* verlassen, da sein Oberzustand `C` eine entsprechende Verbindung besitzt. Zustand `E` wird durch das Ereignis *g* verlassen, da der von Zustand `F` ausgehende Übergang den gesamten AND-Zustand `B` deaktiviert.[21] Hier ist anzumerken, dass weder `E` noch einer seiner Oberzustände eine ausgehende Verbindung besitzt.

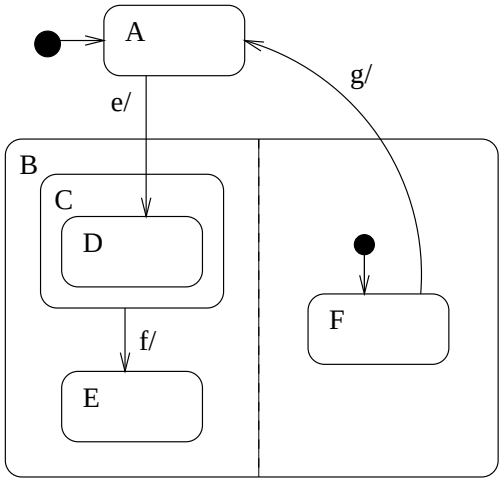


Abbildung 3.7: alle Zustände können verlassen werden

Vollständigkeit

Weitere Fragen, die überprüft werden können, ergeben sich aus dem Umfeld des Defaulteinstiegs. Zunächst einmal sollte jeder OR-Zustand einen Unterzustand als Defaulteinstieg markiert haben. Dies ist jedoch nicht bei jedem Zustand wirklich notwendig: wird der Zustand lediglich durch Verbindungen betreten, die bei einem Unterzustand enden, so würde ein Defaulteinstieg niemals benutzt werden.

[21] Zustand `D` wird selbstverständlich ebenfalls beim Eintreten von *g* verlassen, wenn er noch aktiviert sein sollte.

Eine ähnliche Situation ergibt sich bei den orthogonalen Komponenten eines AND-Zustands. Hier wird der Defaulteinstieg allerdings nicht nur dann notwendig, wenn der AND-Zustand selbst direkt betreten werden kann, sondern auch, wenn es eine Verbindung gibt, die nur in einigen, aber nicht allen orthogonalen Komponenten endet. Durch einen solchen Übergang wird der AND-Zustand und damit alle Partitionen aktiviert – bei Partitionen, die nicht direkt an diesem Übergang beteiligt sind, wird dafür der Defaulteinstieg benutzt. In Abbildung 3.8 wird beispielsweise in Partition `P1` kein Defaulteinstieg benötigt, da jeder Pfeil einen Endzustand in `P1` besitzt. Für den Übergang von `A` auf `B` ist jedoch der Defaulteinstieg in Partition `P2` notwendig.

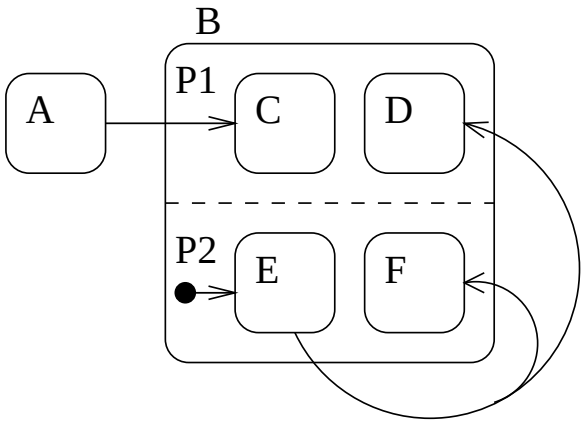


Abbildung 3.8: Notwendigkeit eines Defaulteinstiegs

Der Verifikator sollte also sicherstellen, dass in jedem Zustand und jeder Partition, bei der ein Defaulteinstieg benötigt wird, ein solcher auch vorhanden ist. Dies ist insbesondere beim globalen Wurzelzustand der Fall, weil dadurch der Startzustand des Systems bestimmt wird.

Vollständigkeit lässt sich auch im Zusammenhang mit Ereignissen prüfen. Da erzeugte Ereignisse nicht außerhalb des Statecharts sichtbar sind, kann davon ausgegangen werden, dass sie innerhalb des Statecharts wirken sollen. Wird aber ein Ereignis erzeugt, das an keiner Verbindung verwendet wird, so liegt vermutlich ein Fehler vor; eventuell wurde lediglich vergessen, den Übergang vollständig zu spezifizieren. Umgekehrt kann getestet werden, ob alle Ereignisse, die an mindestens einer Verbindung als Auslöser notiert sind, auch irgendwo erzeugt werden. Hier kann die Antwort allerdings nur dann zuverlässig gegeben werden, wenn die Ereignisse von Elementen des Statecharts erzeugt werden sollen; Ereignisse, die von anderen Programmteilen "hereingereicht" werden, wie etwa Ereignisse, die als Quelltext eingegeben wurden,[22] lassen sich hierdurch allerdings nicht erfassen.

Eine Variante dieser Fragestellung ist die Überprüfung, ob alle Variablen, die durch eine Aktion gesetzt werden, auch irgendwann abgefragt werden. Im Gegensatz zu den Ereignissen kann man bei diesen allerdings nicht davon ausgehen, dass sie notwendigerweise innerhalb des Statecharts selbst wirken; es sind viele Fälle denkbar, in denen die Variablen das Verhalten des Systems außerhalb des modellierten

[22] Diese Möglichkeit wurde bereits auf Seite 66 kurz erwähnt und soll in den Abschnitten 94 und 3.3.2 genauer ausgeführt werden.

Bereichs beeinflussen, beispielsweise innerhalb der Aktionen, die als Quelltext eingegeben wurden. Wird jeder Dialogkomponente eine Variable zugeordnet, wie es auf Seite 69 beschrieben wurde, so entzieht sich diese Frage noch weiter der Überprüfung. Derartige Variablen brauchen weder durch eine Aktion gesetzt werden, da sie über den Benutzerdialog modifiziert werden können, noch brauchen sie in irgendeiner Bedingung getestet werden, da sich ihre beabsichtigte Wirkung auf die Dialogkomponente beschränken kann.

Weitere Konsistenzbedingungen sind für die Zustände des Statecharts denkbar. Ist ein Zustand beispielsweise als transient markiert, so ist es nicht sinnvoll, für diesen eine statische Reaktion anzugeben, da er keinen ganzen Zeitschritt aktiv bleiben wird. Außerdem sollten Endzustände keine ausgehenden Verbindungen haben, da das System anhält, bevor diese benutzt werden könnten.

Races

Als *Race* wird eine Situation beschrieben, in der ein Element gleichzeitig[23] modifiziert und benutzt oder an mehreren Stellen gleichzeitig modifiziert wird.[24] In einem solchen Fall ist das Ergebnis davon abhängig, welche der beiden Aktionen zuerst ausgeführt wird, was aber meistens nicht vom Architekten des Systems beeinflusst werden kann. Man kann hierbei noch zwischen Schreiben/Schreiben-Races und Schreiben/Lesen-Races unterscheiden (write/write- beziehungsweise read/write-races). Abbildung 3.9 zeigt beispielsweise ein Statechart, in dem bei einem Übergang von den Zuständen `(B/D)` auf die Zustände `(C/E)` einer Variablen zwei verschiedene Werte zugewiesen werden. Bei diesem Schreiben/Schreiben-Race ist nicht vorhersagbar, welchen Wert die Variable `X` nach dem Übergang haben wird.

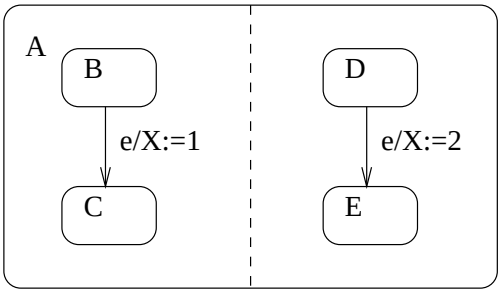


Abbildung 3.9: ein Schreiben/Schreiben-Race

Es wäre daher wünschenswert, solche Schwachstellen im Modell durch den Verifikator identifizieren zu können. Doch dies ist nur in einem begrenzten Maß möglich. Festzustellen, bei welchen Übergängen etwa Variablen, die vom Statechart verwaltet werden, neu gesetzt oder ausgelesen werden ist zwar relativ einfach, aber ob dadurch ein Race entstehen kann, hängt nicht nur davon ab, ob die entsprechenden Bedingungen der Übergänge gleichzeitig erfüllt, sondern auch, ob deren Startzustände gleichzeitig aktiviert sein können. Diese Problemstellung ist offensichtlich der Frage, ob die

[23] „Gleichzeitig“ bedeutet für das diskrete Zeitmodell „in demselben Zeitschritt“.
[24] vgl. [HN96], S. 320 f.

angegebenen Bedingungen das System deterministisch machen, recht ähnlich und daher ebenso schwer zu beantworten.

An dieser Stelle sei aber erwähnt, dass der Begriff Determinismus sich durchaus von Races unterscheidet. Determinismus bezieht sich auf die Auswahl von Übergängen, die von einem aktiven Zustand (und seinen Oberzuständen) ausgehen. Von mehreren möglichen Übergängen wird immer genau einer ausgeführt, und der Architekt des Statecharts kann verschiedene Möglichkeiten geboten bekommen, dafür zu sorgen, dass dies der gewünschte ist.[25] Races treten „über das Statechart verteilt" auf, da sie sich auf eine Menge von Aktionen beziehen, die alle ausgeführt werden. Das Bereinigen von Races ist vergleichsweise aufwendig und kann größere Umstrukturierungen erfordern, um die Zugriffe auf das betroffene Objekt auf mehrere Zeitschritte zu strecken.

3.2.2 Der Simulator

Interaktion mit dem Statechart-Editor

Wie bereits im letzten Abschnitt erwähnt, ist es für einen Verifikator praktisch nicht möglich, etwa die Konsistenz von Bedingungen an ausgehenden Verbindungen oder ähnlich komplexe Fragestellungen zu überprüfen. Andererseits sollten solche Probleme nicht erst beim Einsatz des Programms auffallen. Hier ist die Durchführung einer Simulation unumgänglich.

Der Simulator sollte den Gesamtzustand des Statecharts, also welche Zustände aktiviert sind, sowie die auftretenden Ereignisse und Aktionen verwalten. Darüber hinaus muss er bei Zuständen, bei denen der Einsteig über einen Historykonnektor möglich ist, auch speichern, welche Unterzustände zuletzt aktiv waren. Und schließlich muss er die simulierte Zeit mitzählen, um Timer und ähliche Komponenten korrekt nachvollziehen zu können. Die Art und Weise, wie die Zeit dabei fortschreitet, hängt offensichtlich stark davon ab, ob ein diskretes oder kontinuierliches Zeitmodell gewählt wurde. Zum Vergleich der Modelle siehe Abschnitt 2.8.

Der Simulator kann mit dem Statechart-Editor auf vielen Wegen zusammenarbeiten. Zunächst sollte natürlich in der Ansicht des Editors dargestellt werden, welche Zustände gerade aktiv sind, damit der Benutzer den Globalzustand des Systems überblickt. Dadurch sollten sich grundlegende Fehlkonzeptionen sehr schnell feststellen lassen. Es ist außerdem sinnvoll, sämtliche Ereignisse und Aktionen, die auftreten beziehungsweise ausgeführt werden, anzuzeigen, da der Benutzer dadurch nicht nur den statischen Zustand des Systems erfährt, sondern auch dessen Dynamik nachvollziehen kann.

Das Testen eines Statecharts

Gegenüber dem Verifikator bietet der Simulator die Möglichkeit, die Fragestellungen anzugehen, die für eine automatische Überprüfung zu umfangreich sind. Dazu gehört etwa die in Abschnitt 3.2.1 diskutierte Frage, ob das System durch die angegebenen Bedingungen in allen möglichen Fällen determiniert ist. Auch Races lassen sich durch eine Simulation feststellen. Wichtig ist hierbei, dass der Simulator dadurch nicht in

[25] vgl. Abschnitt 2.4.2

eine Schleife geraten kann, sondern der Benutzer im Notfall jegliche Berechnung abbrechen kann.

Häufig wird Software nur dadurch getestet, dass einige Fälle durchgespielt werden, die möglicherweise zu Fehlern führen könnten. Man kann jedoch nicht sicher sein, dass diese Tests wirklich alle Fehler finden. Es gibt zwei grundsätzliche Methoden, um Software zu testen: Das *white box testing*, bei der besondere Testfälle konstruiert werden, die gezielt auf die Struktur des Systems eingehen, und das *black box testing*, bei dem (mehr oder weniger) willkürliche Tests durchgeführt werden. Die erste Methode lässt sich offensichtlich nur dann anwenden, wenn der Tester Kenntnis von der Programmstruktur hat; sie hat allerdings den Vorteil, dass sich dabei leicht Situationen herbeiführen lassen, die eine hohe Wahrscheinlichkeit haben, Fehler zu finden. Die Klarheit der Struktur von Statecharts ermöglicht es, derartige Situationen recht einfach zu identifizieren und herzustellen.[26] So können bestimmte Zustandsübergänge gezielt angesteuert werden, um festzustellen wo die tatsächliche Struktur des Programms von der geplanten abweicht.

Ein Simulator kann das *white box testing* unterstützen, indem im Statechart gezielt Zustände aktiviert und Ereignisse ausgelöst werden können. So können alle Übergänge der Reihe nach untersucht werden.

Der Dialog zur Simulation

In Abschnitt 69 wurde beschrieben, wie sich Statecharts um einen Benutzerdialog erweitern lassen, indem etwa ein Dialog-Editor angeboten wird. In diesem Fall sollte die Simulation naheliegenderweise mit dem Benutzerdialog arbeiten. Der Tester drückt die Knöpfe, die der Benutzer des Systems auch auswählen könnte, und der Simulator führt die entsprechenden Reaktionen durch. Dies erfordert natürlich die Verbindung des Dialog-Editors beziehungsweise Dialog-Verwalters nicht nur mit dem Statechart-Editor sondern auch mit dem Simulator.

Eine derartige Einbindung des Dialogs eröffnet der Simulation auch eine einfache und elegante Möglichkeit, mit Ereignissen, Aktionen und Bedingungen umzugehen, die vom Benutzer als Quelltext zum Statechart hinzugefügt wurden.[27] Es bietet sich nämlich an, diese Elemente durch einen eigenständigen Dialog zu simulieren; auf diesen wollen wir jedoch erst im Zusammenhang mit der Verwendung von Quelltext in der Simulation (Seite 79) genauer eingehen.

Interessant ist auch die Möglichkeit, den Globalzustand des Systems über den Statechart-Editor beeinflussen zu können. So könnte der Benutzer etwa einige Zustände von Hand aktivieren oder deaktivieren, ohne dass dies auf eventuell umständlichem Weg über die Benutzerschnittstelle geschehen muss. Auch das Eintreten von bestimmten Ereignissen kann so erzwungen werden. Durch solche Modifikationen kann das System natürlich in einen Zustand gebracht werden, der durch die eigentliche Benutzung niemals erreicht werden könnte. Wenn infolge dessen Uneindeutigkeiten auftreten, ist das Statechart nicht notwendigerweise fehlerhaft. Der Simulator sollte sich also merken, ob derartige Änderungen vorgenommen wurden, und im Falle eines Fehlers den Benutzer darauf aufmerksam machen.

[26] vgl. [Hor99a], S. 189 ff.
[27] vgl. Abschnitt 69

Zeit in der Simulation

Die Simulation eines Statecharts kann prinzipiell sowohl in kontinuierlicher oder diskreter Zeit ablaufen. Wie bereits in den Abschnitten 2.8 und 2.5.1 erwähnt wurde, arbeitet Statemate mit einem diskreten Zeitmodell, in dem die Ereignisse genau einen Schritt lang sichtbar sind. Obwohl in verschiedenen Artikeln auch andere Formen beschrieben werden, scheint dieses Modell das zuverlässigste zu sein.[28]

Doch auch im diskreten Zeitmodell gibt es verschiedene Möglichkeiten, wie der Ablauf der Simulation gesteuert werden kann. Es kommt dabei hauptsächlich darauf an, wie viele Schritte das System selbständig durchführt, bis wieder auf eine Eingabe gewartet wird.

Die feinste Aufgliederung wäre eine phasenweise Steuerung, falls die Simulation in Phasen aufgeteilt sein sollte.[29] Eine deratig detailgenaue Betrachtung ist jedoch nicht als erste Stufe der Simulation empfehlenswert, wenn das Statechart auf seine Funktionsweise geprüft werden soll. Sie ist allerdings durchaus sinnvoll, wenn Unterschiede zwischen Konzept und Ausführung des Statecharts auftreten, und festgestellt werden soll, ob diese möglicherweise darauf beruhen, dass einzelne Vorgänge innerhalb eines Übergangs in der falschen Reihenfolge abgearbeitet werden. Auch bei der Analyse von Races kann eine solche Simulation hilfreich sein.

Die nächste Stufe besteht darin, einen einzelnen Übergang auszuführen. Damit kann schrittweise überprüft werden, ob das Statechart so reagiert, wie es vom Architekten vorgesehen wurde. Es kommt nun darauf an, ob ein synchrones oder asynchrones Zeitmodell zugrunde gelegt wurde. Beim synchronen Modell entspricht ein solcher Übergang genau einer Zeiteinheit. Beim asynchronen Modell ist es naheliegend, die Simulation zusätzlich auch in Zeiteinheiten fortschreiten zu lassen, wobei dann gegebenenfalls mehrere Übergänge des Statecharts hintereinander ausgeführt werden, bis die Kontrolle an den Benutzer zurückgegeben wird. Eine solche zeitschrittweise Ausführung der Simulation dürfte für die Analyse des Statecharts am besten geeignet sein.

Letztendlich gibt es natürlich auch die Möglichkeit, die Simulation ununterbrochen laufen zu lassen; dies entspricht am ehesten dem erzeugten Programm. Wenn keine Eingaben des Benutzers vorliegen und keine Übergänge stattfinden, so bleibt das System entsprechend lange in einem Zustand stehen. Eine solche Simulation ist einerseits für den Anfang der Testphase denkbar, wenn noch nicht gezielt nach problematischen Stellen des Statecharts gesucht wird. Andererseits lässt sie sich auch für den Abschluss der Testphase nutzen, um einen Gesamteindruck des modellierten System zu erhalten.

Umgang mit Quelltext in der Simulation

Wie bereits in Abschnitt 69 auf Seite 65 erwähnt, kann ein Statechart-Editor für erweiterte Statecharts auch die Möglichkeit bieten, Quelltextsegmente einzugeben. Dadurch stellt sich die Frage, inwieweit der Simulator auf diese zusätzliche Programmierung eingeht.

[28] vgl. [HN96], S. 294

[29] vgl. die Beschreibung des Simulators von *SCed* in Abschnitt 6.5.2

Im Idealfall sollte dieser Quelltext selbstverständlich komplett ausgeführt werden. Erst dadurch kann die Simulation wirklich sicherstellen, dass das modellierte System auch so funktioniert, wie es geplant ist. Andererseits hieße dies, dass der Simulator beliebigen Quelltext parsen und ausführen können muss – eine Anforderung, deren Umfang die Grenzen eines solchen Werkzeugs sicherlich übersteigt. Entweder wird also der Sprachumfang deutlich eingeschränkt, oder es muss eine andere Möglichkeit gefunden werden, den Quelltext auszuführen.

Eine denkbare Lösung besteht in folgendem Vorgehen: ist das Statechart vollständig konstruiert, so werden die Quelltextsegmente in entsprechender Form als Dateien abgelegt und daraufhin compiliert. Der Simulator wird dann als eigenständiges Programm gestartet, das auf die nun ausführbaren Quelltextsegmente zurückgreifen kann. Im Unterschied zum endgültig erzeugten System soll hierbei jedoch auch noch das Statechart angezeigt werden, um das Verhalten der Simulation daran nachzuvollziehen.

Für den Benutzer einfacher wäre es jedoch, wenn es die Möglichkeit gäbe, diesen Quelltext auszuführen, ohne den Statechart-Editor dafür zuerst zu beenden. In Java ließe sich dies möglicherweise durch das Nachladen von Funktionen zur Laufzeit realisieren.

Dieses Verfahren eignet sich besonders für Aktionen. Auch Bedingungen lassen sich auf diese Weise recht gut in das Statechart integrieren; es muss lediglich darauf geachtet werden, dass diese Funktionen einen Wahrheitswert (*boolean*) zurückliefern.

Komplizierter ist der Umgang mit Quelltext bei Ereignissen. Während im Programm, das den Quelltext enthält, diese Ereignisse automatisch erzeugt werden und das System darauf reagieren kann, muss der Simulator diese Abschnitte gezielt aufrufen und gegebenenfalls die Ereignisse übernehmen oder nachgenerieren. Es ist auch denkbar, dass der Benutzer während der Simulation diese Ereignisse ebenfalls nur simulieren will, anstatt sie vom eigentlichen System erzeugen zu lassen. Möglicherweise handelt es sich um die Meldung einer kritischen Situation im System, die der Benutzer wohl kaum zu Testzwecken herbeiführen möchte.

Der Systemdialog

Eine mögliche Lösung dieses Problems ist es, dem Benutzer einen separaten Dialog zur Verfügung zu stellen, der zur Simulation der externen Ereignisse, Bedingungen und Aktionen dient: der Systemdialog. Dies bietet sich insbesondere an, wenn die Simulation ohnehin über einen Benutzerdialog gesteuert wird, der die im Folgenden erwähnten Dialogelemente bereits zur Verfügung stellt. Selbstverständlich sollten die Elemente mit einem kurzen, prägnanten Namen versehen werden, da sie nicht den kompletten zugehörigen Quelltext anzeigen können.

Ereignisse Ereignisse tauchen für das System plötzlich und gewissermaßen ohne Vorbereitung auf und sind kurz darauf wieder verschwunden. Das Dialogelement, das genau zu diesen Eigenschaften passt, ist der einfache Knopf. Es bietet sich also an, dem Benutzer im Systemdialog für externe Ereignisse Knöpfe anzuzeigen. Durch Drücken eines solchen Knopfes kann er dann das zugehörige externe Ereignis zum gewünschten Zeitpunkt auslösen. Wird die Anzahl der möglichen Ereignisse jedoch groß, kann das Anzeigen eines Knopfes für jedes Ereignis den Systemdialog sehr unübersichtlich

werden lassen. Denkbar ist stattdessen auch, dem Benutzer eine Liste aller möglichen Ereignisse des Statecharts anzuzeigen, und die von ihm in dieser Liste ausgewählten Ereignisse an das Statechart weiterzugeben. Die zweite Methode hat den Vorteil, dass alle externen Ereignisse lediglich ein einziges Feld im Systemdialog beanspruchen; andererseits ist die Bedienung einer solchen Liste in diesem Zusammenhang möglicherweise nicht so leicht zugänglich.

Es sollte dabei allerdings beachtet werden, dass dem Benutzer nicht die Möglichkeit gegeben werden sollte, interne Ereignisse auszulösen, wie etwa das Verlassen eines Zustands (`exit.()`). Im eigentlichen System wird dieses Ereignis nur in bestimmten Situationen auftreten können, nicht zu jedem beliebigen Zeitpunkt. Außerdem setzen interne Ereignisse bereits Aktionen des Statecharts voraus (nämlich in diesem Fall, dass der entsprechende Zustand verlassen wurde), die nicht eingetreten sind, wenn der Benutzer dieses Ereignis gezielt auslöst. Durch diese Einschränkung wird die Anzahl der Ereignisse gegenüber der Gesamtanzahl deutlich reduziert. Besonders komfortabel für den Benutzer wäre es, eine Grenze festlegen zu können, und erst, wenn die Anzahl der Ereignisse diesen Wert übersteigt, diese als Auswahlliste anzuzeigen.

Auch bei externen Ereignissen ist es denkbar, dass sie eine bestimmte Kombination von aktiven Zuständen innerhalb des Statecharts erfordern, bevor sie eintreten können. So wird etwa ein Temparatursensor in einem Wassertank nicht melden, dass das Wasser zu kochen beginnt, wenn sich kein Wasser im Tank befindet. Der Konstrukteur des Statecharts kann solche Zusammenhänge mit Blick auf die Simulation entweder vernachlässigen, da der Benutzer, der den Test durchführt, mit diesem Wissen den Knopf für das entsprechende Ereignis nicht betätigen wird. Wenn der Benutzerdialog aber ohnehin die Möglichkeit bietet, Dialogelemente mittels Aktionen zu deaktivieren,[30] so kann auf diese Weise auch der Systemdialog der Situation angepasst werden. Dies sollte jedoch mit Blick auf den Quelltextexport nur in seltenenen Fällen genutzt werden, da das erzeugte Programm dadurch mit zusätzlichen Aktionen und eventuell sogar weiteren Übergängen ausgestattet wird, die nur in der Simulation eine Bedeutung haben.

Bedingungen Bedingungen sind nach der gebräuchlichen Definition von Statecharts boolesche Ausdrücke.[31] Sie besitzen daher immer einen Wahrheitswert, der entweder *wahr* oder *falsch* ist. Prinzipiell besitzen sie diesen Wert auch dann, wenn sie nicht abgefragt werden, er kann also vom verwendeten Zeitmodell unabhängig sein. Um diese Eigenschaft in einem Dialog nachzuvollziehen, bietet sich eine Checkbox an.[32] Ähnlich wie bei Ereignissen stellt sich hier die Frage, ob sämtliche externe Bedingungen nebeneinander dargestellt werden, oder in einer Liste, mit deren Hilfe der Benutzer den Wahrheitswert der einzelnen Bedingungen auswählen kann. Das Anzeigen aller Bedingungen hat hier jedoch den Vorteil, dass der Benutzer möglicherweise schneller überblickt, welche der Bedingungen momentan falsch oder wahr sind, und so den Gesamtzustand des externen Systems leichter erkennt. Analog zu den Ereignissen ist es auch hier nicht sinnvoll, den Benutzer den Wahrheitswert von internen Bedingun-

[30] vgl. Seite 68
[31] vgl. Abschnitt 2.6
[32] Dieses Dialogelement wurde bereits auf Seite 67 erwähnt.

gen (wie etwa der Abfrage `in?()`, die angibt, ob ein bestimmter Zustand aktiviert ist) wählen zu lassen.

Können Dialogelemente vom Statechart deaktiviert werden, so kann dies auch für im Systemdialog repräsentierte externe Bedingungen geschehen. Hierzu ist jedoch anzumerken, dass in diesem Sinne deaktivierte Bedingungen keinesfalls einen festgelegten Wahrheitswert, wie etwa *falsch*, besitzen; ist eine Checkbox deaktiviert, kann sie lediglich nicht mehr umgeschaltet werden, sondern behält ihren momentanen Zustand bei. Auch dies kann in einigen Fällen sinnvoll sein, beispielsweise wenn es sich um die Anfrage handelt, ob momentan eine längere Prozedur bearbeitet wird, die sich nicht abbrechen lässt, wie etwa das Aufheizen eines Brennofens für Ton.

Es ist allerdings zu bemerken, dass das explizite Ändern des Wahrheitswerts einer externen Bedingung durchaus als externes Ereignis angesehen werden kann. In diesem Fall kann die Bedingung durch ein Zustandspaar innerhalb des Statecharts modelliert und das externe Ereignis wie alle anderen externen Ereignisse verwaltet werden. Andererseits lassen sich zwei externe Ereignisse, die entgegengesetzten Vorgängen entsprechen (wie etwa das Ein- und Ausschalten eines Systems) mittels einer Checkbox zusammenfassen und übersichtlicher verwalten.

Eine andere denkbare Form der Behandlung von externen Bedingungen wäre es, vom Benutzer durch eine Ja-Nein-Anfrage den Wahrheitswert einer Bedingung zu erfragen, wenn dieser benötigt wird. Hierbei ist es nicht notwendigerweise von Vorteil, dem Benutzer den Wahrheitswert bei der letzten Anfrage als Default-Wert vorzuschlagen, da diese Anfrage gegebenenfalls recht lange zurückliegen kann. Es ist jedoch wichtig, dass bei dieser Methode ein intelligenter Algorithmus verwendet wird, damit der Benutzer nicht durch unzählige überflüssige Anfragen belästigt wird. Bei einer Bedingung der Art `extern?(irgendwas) und in?(A)` beispielsweise sollte der Benutzer nicht ununterbrochen den Wahrheitswert der externen Bedingung eingeben müssen, solange die interne Bedingung `in?(A)` nicht erfüllt ist. Die Notwendigkeit der Nachfrage ist jedoch auch bei komplexeren Bedingungen (mit etwas Aufwand) feststellbar.

Aktionen Aktionen werden vom System erzeugt und ausgeführt. Ähnlich wie Ereignisse sind sie nur kurzzeitig zu sehen – nur geht die Infomation hierbei in die entgegengesetzte Richtung: vom Statechart zum Benutzer. Das hierfür bei der Simulation geeignetste Dialogelement (sofern man es als Dialogelement bezeichnen will), ist ein Fenster mit einer Meldung, wie es üblicherweise als Requester oder Popup bezeichnet wird.[33] Dieses Fenster wird zum entsprechenden Moment geöffnet und zeigt dem Benutzer an, welche externe Aktion nun ausgeführt würde. Das System kann sich in diesem Moment in einer Art Wartehaltung befinden, bis der Benutzer dieses Fenster wieder geschlossen hat, oder aber im Hintergrund bereits mit der Simulation fortfahren. Die letztgenannte Möglichkeit kann dazu führen, dass der Benutzer mehrere Meldungen auf dem Bildschirm sieht und deren Reihenfolge nicht mehr feststellen kann. Allerdings dürfte dies nur in den wenigsten Fällen zu einem Problem führen.

Interessant ist die Frage, inwiefern der Benutzer auf ein solches Anzeigen einer Aktion reagieren muss, beziehungsweise ob er anstelle des Simulators den Inhalt der

[33] Diese Elemente wurden bereits im Zusammenhang mit Variablen zum Setzen der Meldung auf Seite 69 erwähnt.

Aktion auszuführen hat. Dies hängt davon ab, ob sich die externe Aktion wieder auf das Statechart auswirkt oder nicht. Wenn dem Benutzer in den Quelltextsegmenten beispielsweise Zugriff auf die Variablen des Statecharts gegeben wird, so können solche Werte auch in Aktionen verändert werden. Wird eine derartige externe Aktion in der Simulation nur gemeldet und nicht wirklich ausgeführt, hat das zur Folge, dass das simulierte Statechart sich vom letztendlich erzeugten Programm unterscheidet.

Es ist allerdings nicht notwendig, den Quelltext der Aktionen in der Simulation auszuführen, um diese Unstimmigkeiten zu vermeiden. Vielmehr sollte darauf geachtet werden, dass in den Quelltext von Aktionen keine Befehle eintragen werden, die das Statechart beeinflussen, sondern nur solche, die sich auf den Rest des Systems auswirken. Müssen Variablen oder Bedingungen in ihrem Wert geändert werden, so sollte dies nicht Teil der externen Aktion sein, sondern als zusätzliche interne Aktion spezifiziert werden. Auf diese Weise entspricht die Simulation dem erzeugten Quelltext, ohne dass der Benutzer externe Aktionen selbst ausführen müsste.

3.3 Automatismen

3.3.1 Der Optimierer

Eine wünschenswerte Funktion im täglichen Entwurf ist es, vorhandene manuelle Entwürfe automatisch zu optimieren. Zuerst muss man hier definieren, was man unter Optimieren versteht. In Abgrenzung zur automatischen Generierung soll die Optimierung ein bereits vorhandenes Statechart in ein funktionsäquivalentes Statechart überführen, welches bezüglich eines Optimierkriteriums besser oder gar optimal ist. Optimierkriterien können sein:

- Anzahl der Zustände
- Länge der Kanten
- Fläche des Statecharts
- Tiefe des Statecharts
- Anzahl der verwendeten Variablen

Weitere Kriterien sind denkbar. Zu Beachten ist hier eine Unterscheidung zwischen Optimierungen des logischen Aufbaus und Optimierungen des grafischen Layouts.

Optimierung des grafischen Layouts

Unter der Optimierung des grafischen Layouts verstehen wir die Anordnung der Komponenten des Statecharts auf minimaler Fläche. Da jedoch die Elemente des Statecharts per definition erst einmal gar keine Größe haben müssen, sondern nur Repräsentationen eines logischen Modells sind, muß man Ihnen zu Beginn einer solchen Optimierung minimale Dimensionen zuweisen. Dann wiederum ist dieses Problem jedoch weitgehend äquivalent zu dem Problem des automatischen Schaltungsentwurfs

und es gibt damit ein gut erforschtes Gebiet, dessen Methoden und Ergebnisse hierauf angewendet werden können.[34] Ein Beispiel zeigt Abbildung 3.10.

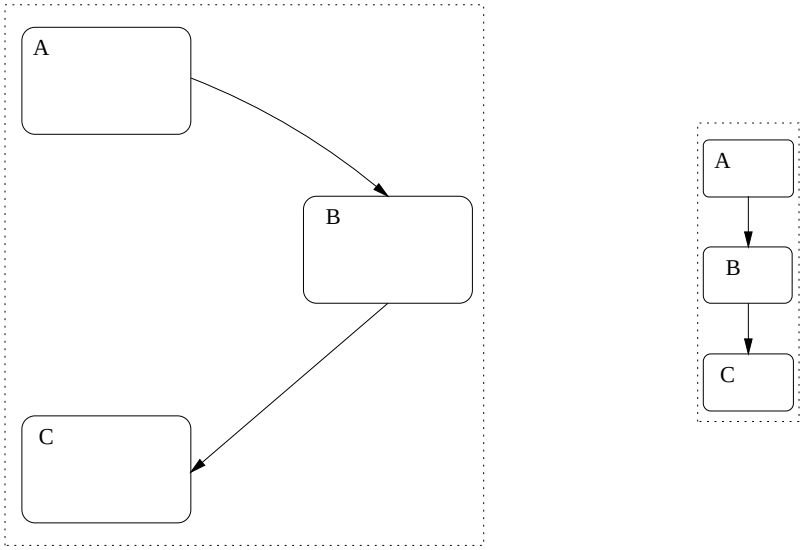


Abbildung 3.10: Optimierung des grafischen Layouts

Optimierung des logischen Aufbaus

Die Optimierung des logischen Aufbaus eines Statecharts soll ein – in gewisser Hinsicht – einfacheres Statechart generieren, welches die gleiche Funktion erfüllt. Abhängig vom Optimierkriterium sind folgende Überlegungen hilfreich:

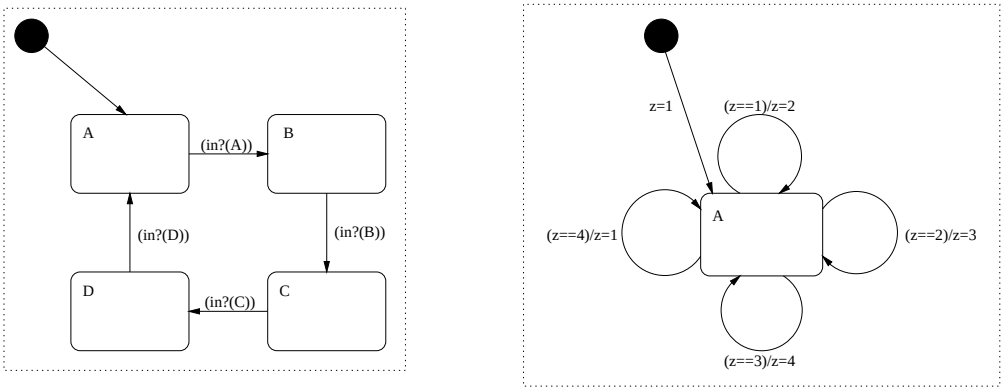


Abbildung 3.11: Optimierung nach Anzahl der Zustände

Optimierung nach Anzahl der Zustände In Anlehnung an endliche Automaten kann man hier versuchen, Zustände zu verschmelzen. Aufgrund der Möglichkeit, Va-

[34] siehe hierzu auch [Brü93] und [Len90]

riablen in einem Statechart zu testen und zu setzen, kann man jedoch auch jedes Statechart in ein funktionsäquivalentes Statechart mit nur einem einzigen Zustand überführen. Man bildet hierzu eine Abzählung aller möglichen Zustandskombinationen. Diese ist bei einem endlichen Statechart auch immer endlich. Nun führe man eine neue Variable ein, welche diese Zustandskombination repräsentiert. Alle Bedingungen, die nun Zustände abfragen, werden umgeschrieben auf Bedingungen, die diese Variable vergleichen. Ein Beispiel zeigt Abbildung 3.11. Diese Optimierung ist im allgemeinen jedoch wenig vorteilhaft, da der Sinn des Statecharts gerade in der symbolischen Repräsentation von Zuständen liegt.

Eine weitere Möglichkeit ist das Erkennen von nebenläufigen Eigenschaften, die dann durch einen AND-Zustand parallelisiert werden können.

Beispiel. Die Zustände sollen sich die Attribute eines Schrifttyps merken. Diese können sein *bold (b)*, *italic (i)* und *underline (u)*, sowie beliebige Kombinationen daraus. Dies ergibt in sequentieller Repräsentation 2^3 Zustände. Parallel reichen für jeden Schrifttyp 2, also insgesamt $2 \cdot 3$ Zustände. Ein Optimierer könnte nun versuchen, solche Zusammenhänge zu erkennen und zu parallelisieren. Abbildung 3.12 zeigt dieses Beispiel, wobei auf die Beschriftung der Kanten verzichtet wurde.

Offensichtlich werden durch diese Art der Optimierung auch Kanten eingespart. Außerdem sind Bedingungen, die sich auf ein Attribut beziehen, jetzt leichter zu formulieren; andererseits werden Bedingungen, die sich auf eine Zustandskombination beziehen, schwieriger zu formulieren.

Diese Art der Optimierung scheint algorithmisch sehr aufwendig zu sein, da vor allem das Erkennen von solchen nebenläufigen Zuständen schwierig ist. Ein Ansatz, den man hierfür überprüfen könnte, ist die Frage, ob die topologische Struktur einen Hinweis auf eine solche Nebenläufigkeit bietet, und ob man diese immer Eindeutig bestimmen kann.

Optimierung nach Anzahl der Variablen Umgekehrt zur Optimierung nach Anzahl der Zustände ist es hier möglich, jede Variable, die in einem Statechart nur einen endlichen Wertebereich annehmen kann, durch je einen Zustand für jeden möglichen Wert zu repräsentieren. Ein Beispiel zeigt Abbildung 3.13. Man beachte, dass die Werte der Variablen nicht aufeinander folgen müssen.

Diese Art der Optimierung ist nur dann sinnvoll, wenn die verschiedenen Werte der ursprünglichen Variable auch unterschiedlich im Sinne von „unterschiedlichen Zuständen“ behandelt werden.

Optimierung nach Tiefe des Statecharts Für eine Optimierung der Tiefe betrachten wir zuerst komplexe Zustände ohne History.

Hierbei lassen sich dann OR-Zustände einfach auflösen, indem man nur die inneren Basic-Zustände benutzt. Alle Kanten, die Defaulteinstiege nutzen, müssen dann direkt auf untere Zustände zeigen, und Bedingungen, die auf Oberzustände testen, müssen dann alle Unterzustände testen.

AND-Zustände sind hierbei ein wenig komplexer. Man betrachte auch hier wieder nur die Basic-Zustände und bilde für alle möglichen Kombinationen neue Zustände und arbeite mit diesen. Die Kanten müssen entsprechend den Kombinationen gezogen

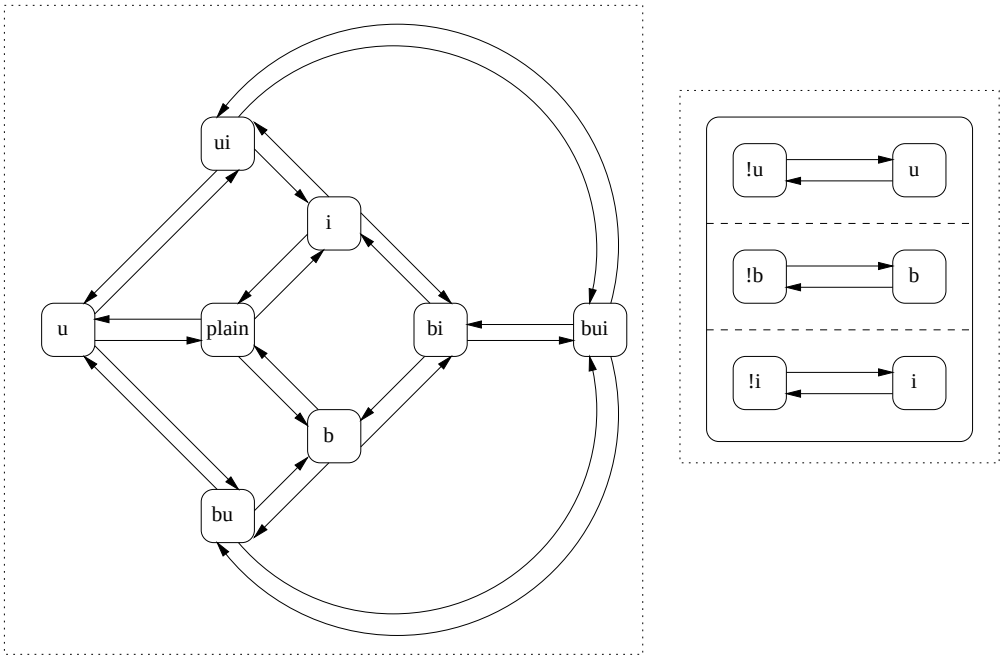


Abbildung 3.12: Optimierung durch Parallelisierung

werden. Das Beispiel in Abbildung 3.12 zeigt den prinzipiellen Vorgang in umgekehrter Richtung.

Wenn nun Historyeinstiege ins Spiel kommen, müssen für jeden komplexen Zustand entsprechend die History-Informationen, zum Beispiel in einer Variablen, gespeichert werden. Die Kanten müssen dann gegebenenfalls abhängig von dem Wert dieser Variablen verzweigen.

Bemerkung Einer der größten Vorteile von Statecharts ist die Übersichtlichkeit und die leichte Durchschaubarkeit von Entwürfen. Gerade diese kann jedoch sehr leiden, wenn man ein Statechart nach obigen Kriterien optimiert. Ob eine Optimierung dennoch sinnvoll sein kann, muss dann im Einzelfall entschieden werden. Es lassen sich keine allgemeingültigen Regeln für die Abwägung zwischen Übersichtlichkeit und Optimierung angeben.

3.3.2 Der Quelltext-Generator

Einer der Vorteile von Statecharts ist deren streng logische Konstruktion. Das Umsetzen eines Diagramms in Quelltext ist daher sehr einfach – und sogar automatisch möglich. Aus dieser Überlegung bietet es sich an, einen Quelltext-Generator in die Sammlung der Werkzeuge aufzunehmen.

Benutzergeschriebener Quelltext

Das Verhalten des modellierten Systems wird durch das Statechart vollständig beschrieben, die eigentlichen Reaktionen des Systems können jedoch nur durch Quell-

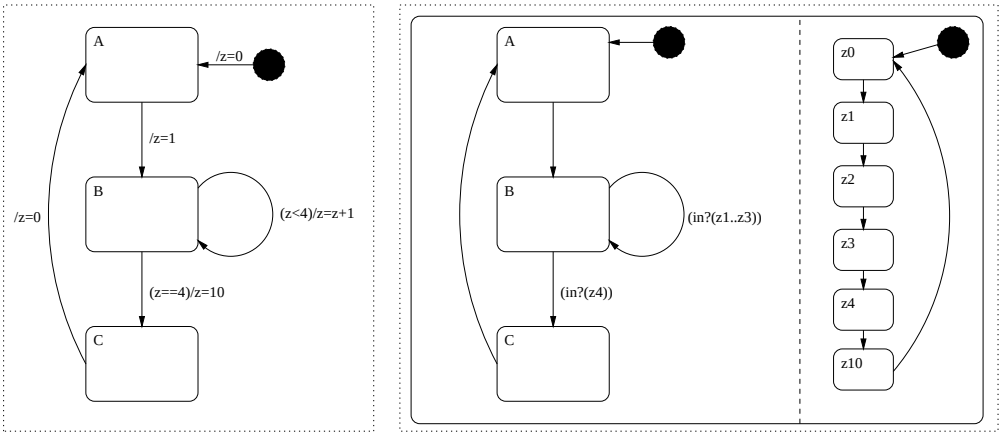


Abbildung 3.13: Optimierung nach Anzahl der Variablen

textsegmente programmiert werden. Diese sind natürlich in Aktionen, aber auch in Ereignissen und Bedingungen denkbar. Im Zusammenhang mit der Simulation wurde darauf bereits in Abschnitt 94 auf Seite 79 eingegangen.

Auf Seite 83 wurde bereits davor gewarnt, der Versuchung zu erliegen, durch aufwendig programmierte externe Aktionen die Stuktur und Kontrolle des Statecharts zu umgehen. Insbesondere sind in Aktionen Konditionalstrukturen oder Bezüge auf Zustände des Statecharts zu meiden: diese sollten innerhalb des Statecharts modelliert werden. Es ist daher nur sinnvoll, den Benutzer sozusagen atomare Aktionen in einer Programmiersprache ergänzen zu lassen.

Während der Quelltext, der in Aktionen angegeben wird, relativ einfach mit dem Statechart verbunden werden kann, da die Anweisungsfolge wie bei direkter Programmierung linear ausgeführt wird, ist das Programmieren von Ereignissen und Bedingungen etwas komplizierter.

Atomare Bedingungen sollten als Funktionen programmiert werden, die einen Wahrheitswert zurückliefern. Dieser kann dann direkt weiterverwendet werden, um Übergänge zu ermöglichen oder zu verhindern. Problematisch ist hierbei die Angabe, welche Objekte dieser Funktion übergeben werden sollen. Einfacher zu verwalten wären global gehaltene Objekte, die selbstverständlich nicht übergeben zu werden brauchen, da jede Funktion direkt darauf zugreifen kann. Dies führt jedoch sehr schnell zu einer monolithischen Programmstruktur, die möglicht vermieden werden sollte.

Sehr wichtig ist es jedoch, darauf zu achten, dass die Auswertung einer Bedingung nicht zu Seiteneffekten führt. Dies wird bereits in anderen Arbeiten mehrfach erwähnt.[35]

Ereigniss-Konzepte Am kompliziertesten dürfte es sein, Ereignisse frei zu programmieren. Dafür muss nicht nur festgelegt werden, in welcher Weise diese zu formulieren sind, sondern auch, wie sie in den erzeugten Quelltext des Systems eingebunden werden können.

[35] vgl. beispielsweise [Rum99], S. 291, oder [Schu99], S. 184.

Eine denkbare Variante ist es, Ereignisse wie Interrupts (oder in Java: Exceptions) zu programmieren, die bei Eintreten an das Statechart weitergegeben und von diesem abgearbeitet werden. In diesem Fall muss das komplette Programm als Teil des Statecharts programmiert werden. Es wäre möglich, einige größere Funktionen separat zu halten und mittels externer Aktionen aufzurufen, hierfür müsste der Programmierer jedoch den automatisch erzeugten Quelltext nachträglich entsprechend erweitern.

Wird das Statechart für das erzeugte Programm andererseits lediglich als Modul konzipiert, so kann es von einem umfangreicheren Hauptprogramm genutzt werden. Diese Variante hat den Vorteil, dass der Quellcode, der innerhalb des Statecharts gehalten wird, auf das eigentliche Programm umgelagert und der Umfang des Statecharts damit reduziert werden kann. Hierbei wird das Statechart, ähnlich wie beispielsweise Fenster oder andere Widgets in Java, vom Hauptprogramm erzeugt, läuft dann aber unabhängig von diesem weiter. Die Kommunikation vom Hauptprogramm zum Statechart wird über direkte Funktionsaufrufe, die umgekehrte Richtung über das (in Java übliche) Listener-Konzept stattfinden. Ereignisse können nun im Hauptprogramm erkannt und an das Statechart übergeben werden; dies vereinfacht den Überblick über das endgültige Programm und erleichtert damit die Einbindung eines Statecharts.

Eigenschaften des erzeugten Programms

So übersichtlich Statecharts auch sind, die Umsetzung in Quelltext ist nicht notwendigerweise annähernd leicht lesbar. Wenn er zudem ohne das ursprüngliche Statechart betrachtet wird, dürfte er vollends unverständlich sein. Daher sollte beim Erzeugen des Quelltextes darauf geachtet werden, dass dem Programmierer, der die Software erweitern will, geeignete Methoden zur Vefügung gestellt werden, mit denen er arbeiten kann, ohne den automatisch erzeugten Code im Detail nachvollziehen zu müssen.

Einerseits gilt dies für die frei programmierbaren Abschnitte, die innerhalb des Statecharts gespeichert werden, nämlich den externen Aktionen und Bedingungen. Bei den Aktionen ist es jedoch nicht notwendig, auf sämtliche Teile des Statecharts Einfluss nehmen zu können; wie es auf Seite 83 bereits erwähnt wurde, sollten derartige Aktionen als interne Aktionen programmiert werden. Analoges gilt für die Bedingungen: wenn sie Eigenschaften testen, deren Wahrheitswert sich innerhalb des Statecharts ermitteln lässt, sollten sie möglichst als interne Bedingungen programmiert werden.

Die Ereignisse andererseits lassen sich aus dem Statechart auslagern, wenn dieses lediglich als Modul betrachtet wird, wie es im letzten Abschnitt beschrieben wurde. Hierfür benötigt das Statechart selbstverständlich eine zugängliche Methode, die ein solches Ereignis an das Statechart weiterreicht. Die einfachste Art, dem Statechart zu sagen, dass ein bestimmtes Ereignis aufgetreten ist, dürfte im Übergeben eines Strings liegen. Beim Konstruieren des Statecharts wird für jedes derartige Ereignis ein Text festgelegt, durch den sie sich unterscheiden lassen, und das ausführbare Programm verwendet dieselben Texte auch außerhalb des Statechart-Moduls.

Die Struktur des erzeugten Quelltexts ließe sich auch feiner unterteilen, wenn im Programmpaket ein Programmstruktur-Editor enthalten ist, wie er in Abschnitt 3.1.3 beschrieben wurde. Die dort erwähnten Module können dann als einzelne Klassen oder Instanzen implementiert werden, wodurch das Programm in fast selbständige Einheiten zerlegt würde. Ohne einen derartigen Editor ist diese Unterteilung kaum durchführbar, da die gewünschte Struktur nicht automatisch festgestellt werden kann.

Simulation und Quelltext

Es dürfte recht offensichtlich sein, dass die Anforderungen an einen Simulator und an den erzeugten Quelltext relativ ähnlich sind. Beide erfüllen das Statechart mit Leben, indem sie den Spezifikationen folgend Übergänge und Aktionen ausführen. Wenn dem Benutzer neben dem Editor beide Werkzeuge zur Vefügung gestellt werden, ist es also naheliegend, den Quelltextexport an die Simulation anzulehnen.

Das erzeugte Programm unterscheidet sich in zwei Punkten von der Simulation: die Art und Weise der Ausführung kann nicht beeinflusst werden, da sie kontinuierlich ablaufen sollte,[36] und die externen Ereignisse, Bedingungen und Aktionen werden nicht über einen Systemdialog an das Statechart weitergegeben. Für externe Aktionen und Bedingungen ist die Benutzung von Quelltextabschnitten empfehlenswert, wie sie bereits auf Seite 65 beschrieben wurde. Es wird dadurch nicht notwendig, den erzeugten Programmcode nachträglich um diese Methoden zu erweitern. Externe Ereignisse lassen sich in den beiden auf Seite 87 erwähnten Arten einbinden.

[36] vgl. die Beschreibung der Zeit in der Simulation auf Seite 79

Teil II

Grundlagen von *SCed*

Höchste Vollendung scheint unvollkommen zu sein,
doch ihre Anwendung ist unerschöpflich.
(Lao-Tse, Tao-Te-Ting, Kapitel 45)

Kapitel 4

Die implementierte Fassung

4.1 Einleitung

Die Form, in der Statecharts im vorliegenden Programmpaket implementiert sind, unterscheidet sich etwas von den in Kapitel 2 beschriebenden Fassungen. In diesem Kapitel soll daher unser Ansatz vorgestellt und die Gründe für unsere Entscheidungen dargelegt werden.

In vielen Punkten haben wir den in der Praxis bewährten Ansatz des Programms Statemate übernommen. Insbesondere benutzen auch wir eine diskrete Zeitskala, und Ereignisse werden erst im nächsten Zeitschritt für das System sichtbar. Diese Variante hat unter anderem den deutlichen Vorteil, dass es nicht auf die Reihenfolge der Abarbeitung von Übergängen oder Aktionen ankommt, da sie sich innerhalb eines Zeitschrittes nicht gegenseitig beeinflussen.

Dieses Kapitel beschreibt nicht, welche der in Kapitel 3 beschriebenen Werkzeuge implementiert und auf welche Weise diese umgesetzt wurden. Für eine solche Darstellung sei auf das Benutzerhandbuch in Kapitel 5 beziehungsweise die Beschreibung der Funktionsweise in Kapitel 6 verwiesen.

4.2 Die Grundelemente

4.2.1 Zustände

Die Eigenschaften unserer Zustände entsprechen größtenteils denen der in Statemate implementierten Zustände. Es gibt jedoch ein paar Unterschiede, die im folgenden beschrieben werden.

Bedingungen am Defaulteinstieg

Wie in Abschnitt 2.4.1 erwähnt, lassen Harel und Namaad zusätzliche Ereignisse und Bedingungen an Verbindungen zu, die von einem Defaulteinstiegs-Konnektor ausgehen. Dies kann jedoch bei komplexeren Statecharts sehr schnell dazu führen, dass der Betrachter nicht mehr erkennt, welche Einschränkungen tatsächlich für einen Übergang gelten. Insbesondere in Kombination mit Historyeinstiegen verschleiert dies die Funktionsweise des Statecharts, wie beispielsweise in Abbildung 4.1 dargestellt.

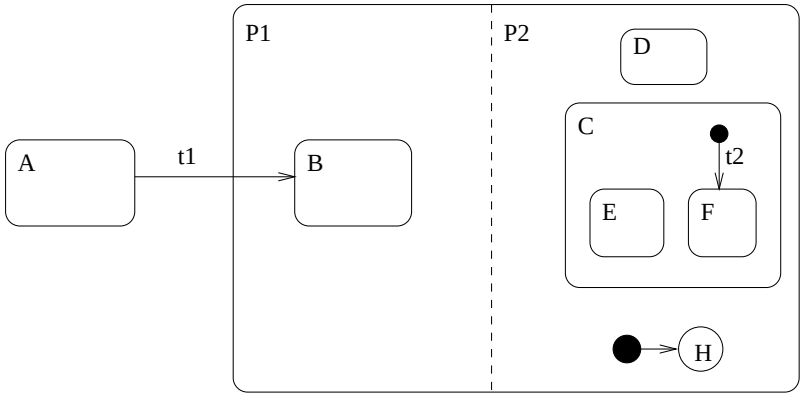


Abbildung 4.1: Konstruktion mit Bedingungen am Defaulteinstieg

Angenommen, Zustand A ist aktiviert und der Übergang t_1 möglich, der Übergang t_2 jedoch nicht. Dann hängt es davon ab, ob die History von Partition P2 auf C oder D verweist; für die History D ist der Übergang von A auf B erlaubt, für die History C nicht, da dieser Zustand nicht über seinen Defaulteinstieg betreten werden kann. Problematisch ist hierbei einerseits die Verteilung der ausschlaggebenden Elemente über weit auseinanderliegende Bereiche des Statecharts, andererseits auch die Tatsache, dass der Wert der History sich nicht nur auf den direkten Unterzustand sondern auch parallele Zustände auswirkt. Insbesondere sind die Reaktionen des modellierten Systems in Abhängigkeit der History so unterschiedlich, dass es nicht ohne weiteres möglich ist, diese ohne eine Simulation einzuschätzen.

Wir haben uns daher entschieden, beim Defaulteinstieg weder weitere Ereignisse noch Bedingungen zuzulassen, da damit diese Unübersichtlichkeit ausgeschlossen wird.

Darüber hinaus können in *SCed* nur direkte Unterzustände als Defaulteinstieg markiert werden. Dass sich ein Defaulteinstieg nicht über mehrere Ebenen der Zustandshierarchie erstrecken darf, ist keine wirkliche Einschränkung, da ebensogut in jedem Zustand ein eigener Defaulteinstieg definiert werden kann; dies trägt außerdem zur Übersichtlichkeit der Konstruktion bei. Doch auch ein Defaulteinstieg, der auf einen Konnektor verweist, kann die Klarheit des Statecharts beeinträchtigen. Hierzu sei auf das in Abschnitt 2.3.2 auf Seite 46 beschriebene Problem im Verbindung mit einem History-Konnektor oder auf Abbildung 2.9 verwiesen, die Schwierigkeiten mit zusammengesetzten Verbindungen darstellt.

Parametrisierung

Parametrisierte Zustände sind solche, die im Statechart nur einmal spezifiziert sind, zur Laufzeit des Systems aber wie mehrere verschiedene Zustände gehandhabt werden. Es gibt hierfür verschiedene Konzepte, die in Abschnitt 2.2.3 auf Seite 39 vorgestellt wurden.

Wir haben uns für eine Variante entschieden, die der von Horrocks vorgeschlagenen ähnelt: durch die Einführung von Variablen in das Statechart ist es möglich, das Verhalten eines Zustands zu beeinflussen und etwa bei jedem Aufruf eine andere Feh-

lermeldung auszugeben. Auf diese Weise lässt sich auch die von Harel vorgeschlagene OR-Parametrisierung nachmodellieren, da diese Form der Parametrisierung immer einer Art Zähler ähnlich ist.

Die AND-Parametrisierung wollen wir jedoch nicht unterstützen. Im Folgenden soll diese Einschränkung erklärt werden.

Das Programm *SCed* verwaltet ein einziges Statechart mit einem globalen Namensraum. Diese Entscheidung wollen wir zunächst begründen. Ein globaler Namensraum ist notwendig, da mehrfach auftretende Namen von Zuständen zu uneindeutigen Bezügen führen können, wie die Beispiele in Abbildung 4.2 zeigen. Im ersten Fall ist es nicht klar, auf welchen der Zustände mit dem Namen `ON` sich die Bedingung am Übergang in Partition *P1* bezieht, da beide auf derselben Ebene der Zustandshierarchie liegen. In Fall (b) wäre es sogar denkbar, dass sie sich auf die (oberhalb liegende) Partition *P2* bezieht, so dass die Lokalisierung des Namensraumes hier eine Einschränkung der Möglichkeiten bedeuten würde.[1]

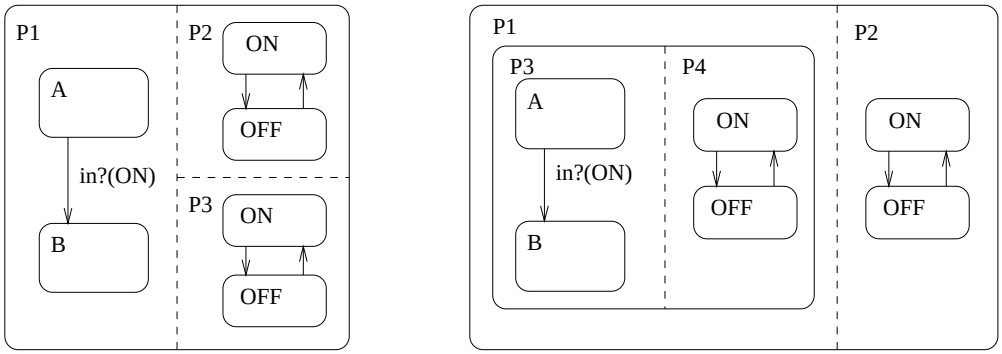


Abbildung 4.2: Probleme bei lokalem Namensraum für Zustände

Auch für Variablen, Ereignisse und Aktionen gilt ein solcher globaler Namensraum. Das Binden dieser Objekte an einzelne Zustände würde zu Schwierigkeiten bei der Behandlung der Übergänge führen. So stellt sich die Frage, auf welchen Zustand sich die Angaben von Ereignissen und Aktionen eines Übergangs beziehen. Dadurch, dass Übergänge bei mehreren Zuständen beginnen und enden können, ist es nicht möglich, die Start- und Endzustände hierfür zu benutzen. Und obwohl es genau einen Zustand im Statechart gibt, der als erster sämtliche Start- und Endzustände eines Übergangs enthält (*Scope*), ist es auch nicht konsistent, die Angaben auf diesen Zustand zu beziehen, wie Abbildung 4.3 zeigt. Hier gibt es einen Zustand, der eine Fehlermeldung anzeigt; der Text dieser Fehlermeldung wird in der Variablen *ERROR* gespeichert und auf diese Weise dem Zustand übergeben. Offensichtlich ist es hier nicht sinnvoll, diese Variable im Scope einer der beiden Übergänge zu halten, da eventuell noch weitere Verbindungen bei diesem Fehlerzustand enden, wodurch die Variable in höherliegende Zustände der Hierarchie verlagert werden müsste. Obwohl es in diesem Fall schön

[1] Die globale Eindeutigkeit wird im Programm durch eine fortlaufende Nummerierung sichergestellt. Es ist daher nicht notwendig, dass der Benutzer sich für jeden weiteren Zustand einen neuen Namen überlegt.

wäre, die Textvariable im Fehlerzustand selbst zu verwalten, würde dies zur bereits erwähnten Inkonsistenz bei der Verwaltung von 1-N Verbindungen führen.

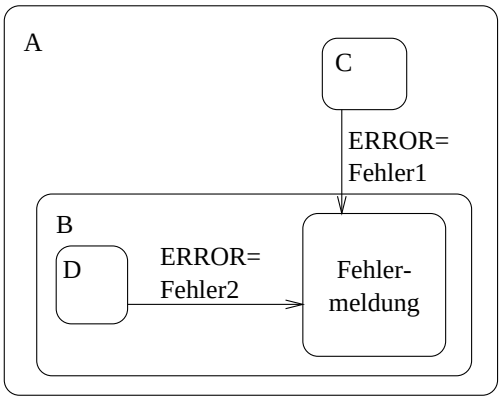


Abbildung 4.3: Inkonsistenz bei lokalem Namensraum für Variablen

Es ist also durchaus angebracht, einen globalen Namensraum anzunehmen. Dies wirft jedoch in Verbindung mit einer AND-Parametrisierung erhebliche Probleme auf. So müssen die Zustände, Ereignisse und Aktionen einer bestimmten Instanz des parametrisierten Zustands zugeordnet werden können, um im Namensraum nicht zu Konflikten zu führen. Insbesondere, wenn man geschachtelte AND-Parametrisierungen erlaubt, wird dadurch eine relativ aufwendige Kennzeichnung notwendig.

Außerdem stellt sich die Frage, ob und wie auf die Unterzustände eines mehrfach auftretenden Zustands zugegriffen beziehungsweise auf dort ausgelöste Ereignisse reagiert werden kann. Prinzipiell sind diese ja wegen des globalen Namensraumes im gesamten Statechart sichtbar und können etwa durch Bedingungen abgefragt werden. Da aber die Anzahl der Instanzen nicht notwendigerweise festgelegt ist, kann keine bestimmte Instanz angesprochen werden. Es kann sogar vorkommen, dass keine einzige Instanz dieses Zustands existiert. Hier wäre etwa eine mehrwertie Logik notwendig, um diesen Fall abzufangen. Andererseits ist es auch nicht unproblematisch, diese Anfragen mit Existenz- oder Allquantoren zu versehen und dadurch das Verhalten des gesamten Systems entweder durch eine einzige Instanz beeinflussen zu lassen oder darauf zu warten, dass alle Instanzen eine bestimmte Bedingung erfüllen. In diesem Zusammenhang ist es zusätzlich notwendig festzulegen, wie das System derartige Anfragen modifiziert, wenn neue Instanzen hinzukommen oder alte aufhören zu existieren.

In Statemate werden diese Schwierigkeiten durch die Einführung von generischen Charts gelöst, die bereits in Abschnitt 2.2.3 vorgestellt wurden. Diese Charts sind vom globalen Namensraum des Statecharts getrennt und können mit diesem nur über Parameter kommunizieren. Eine solche Auskopplung von Teilen des Statecharts ist zwar denkbar, würde jedoch den Umfang der Implementierung deutlich erhöhen. Da diese Problematik nicht den zentralen Teil unserer Arbeit darstellt, wird die AND-Parametrisierung deshalb nicht unterstützt. Ansatzweise kann sie jedoch (mit einer festgelegten Anzahl an orthogonalen Instanzen) nachmodelliert werden, indem der betroffene Zustand im Statechart mehrfach kopiert wird.

4.2.2 Konnektoren

In Statemate werden, wie bereits in Abschnitt 2.3 erwähnt, sieben Formen von Konnektoren unterschieden, die in zwei Gruppen eingeteilt werden können. Der hauptsächliche Vorteil der OR-Konnektoren ist von graphischer Art, da das Verhalten in jedem Fall auch durch mehrere alternative Übergänge modelliert werden kann. Die Unterschiede zwischen den drei Arten von OR-Konnektoren sind demzufolge auch nicht logischer Natur – die Einführung der drei Typen wird von Harel und Politi wie folgt begründet:[2]

> „Die einzelnen Konnektoren sind dazu gedacht, die Unterscheidung zwischen verschiedenen Verhaltensweisen optisch hervorzuheben: ein C-Konnektor [condition] deutet eine Verzweigung auf Grund von Bedingungen an, ein S-Konnektor [Switch-Konnektor] verzweigt durch Ereignisse, und Vereinigungs-Konnektoren werden für alle übrigen Fälle benutzt."

Wir haben uns daher dafür entschieden, diese Konnektoren zu einem Typ zusammenzufassen, dem Universal-Konnektor.

AND-Konnektoren bieten den Vorteil, dass sie sich in natürlicher Weise auf mehrere Partitionen eines AND-Zustands gleichzeitig beziehen können. Wir haben uns wegen der parallelen Natur dieser Übergänge dafür entschieden, diese Funktionalität nicht zu den Konnektoren, sondern zu den Verbindungen zu zählen. Deshalb haben wir neben den üblichen 1-1-Übergängen noch zwei weitere Typen definiert: N-1- und 1-N-Übergänge, die von mehreren Startzuständen ausgehen beziehungsweise bei mehreren Endzuständen aufhören. Bei der Simulation und Verifikation ergeben sich dadurch Vorteile gegenüber der Modellierung mit mehreren einzelnen Verbindungen und Konnektoren. So wird beim Verlassen des AND-Zustands `S` mittels eines N-1-Übergangs nur ein Ereignis `exit.(S)` erzeugt, und das Benutzen eines 1-N-Übergangs ist von Nichtdeterminismus leichter zu unterscheiden, als das bei einer Vielzahl von Verbindungen der Fall wäre. Die Notwendigkeit eines N-N-Übergangs ist allerdings nicht gegeben, da ein solcher Übergang sich mit Hilfe eines Konnektors aus einem N-1- und einem 1-N-Übergang äquivalent konstruieren lässt.

Es sei in diesem Zusammenhang erwähnt, dass wir nur N-1-Übergänge zulassen, die von Zuständen ausgehen, nicht etwa von Konnektoren. Dies ist jedoch keine Einschränkung der Konstruktionsmöglichkeiten.

Der in der [Har87b] beschriebene Historyeinstieg wird in Statemate als Konnektor implementiert;[3] dieser Entscheidung haben wir uns angeschlossen, da dadurch die Möglichkeit entsteht, den Historyeinstieg mit ausgehenden Verbindungen zu versehen.

Da der in Statemate angebotene Diagramm-Konnektor keine logisch notwendige Funktion besitzt, haben wir diesen Konnektor nicht implementiert.

4.2.3 Ereignisse, Aktionen und Bedingungen

Die Verwendung von Ereignissen und Aktionen ist nicht einheitlich. Wir entscheiden uns für folgende Verwendungsweise:

[2] vgl. [HP98b], S. 67
[3] vgl. [HP98b], S. 71

Die Ereignisse

Ereignisse haben als einzige Eigenschaft ihre Unterscheidbarkeit.[4] Ihre Verwendung liegt allein in der notwendigen Kombination mit einer Transition, denn jede Transition muss ein Ereignis als auslösendes Moment besitzen. Ereignisse können nicht direkt erzeugt werden, es können stattdessen Aktionen ausgeführt werden, welche ein Ereignis generieren (wir bezeichnen derartige Aktionen als *Throw-Aktionen*). Insofern besitzen alle Objekte, welche in Harels Statechart-Semantik Ereignisse generieren können, die Möglichkeit, Aktionen zu erzeugen.[5]

Beispiel.

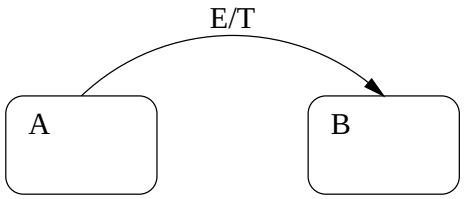


Abbildung 4.4: Beispiel zu Aktionen und Ereignissen

Betrachten wir das Beispiel in Abbildung 4.4. Zustand `A` sei aktiv und im vorherigen Schritt wurde das Ereignis *E* generiert. Die Transition wird nun also ausgeführt. Zustand `A` und Zustand `B` generieren Aktionen. Voreingestellt generiert Zustand `A` die Aktion `throw!(exit.(A))`, Zustand `B` die Aktion `throw!(enter.(B))`, und der Übergang generiert die Aktion `T`.

Weiterhin ist zu beachten, dass wir uns auf genau ein Ereignis je Transition beschränken, da es die besondere Stellung des Ereignisses als auslösendes Moment *(Trigger)* einer Transition hervorhebt.

Warum nur ein Ereignis? Wollte man mehr als ein Ereignis als Auslöser für eine Transition definieren, muss man sich zuerst überlegen, ob die Transition bei gleichzeitigem Eintreten durchgeführt werden soll, oder sobald eines eintritt. Die Möglichkeit der *Und*-Verknüpfung wollen wir ausschließen. Dies muss in einem Statechart im allgemeinen anders modelliert werden. Übergänge sollen durch ein Ereignis ausglöst werden und nicht durch die Kombination mehrerer. Denkbar sind hier Abfragen von Zuständen in Bedingungen.

Die *Oder*-Verknüpfung lässt sich in unserem Modell nachbilden, indem man einfach mehrere parallele Kanten zeichnet.

Beispiel.

Will man die Transition in Abbildung 4.5 modellieren, so kann man diese in unserem Modell wie in Abbildung 4.6 darstellen. Noch eleganter kann man die Verknüpfung mit Konnektoren lösen, da dann die Aktion nur an einer Kante steht (siehe Abbildung 4.7).

[4] Dieses Prinzip wird auch im UML-Werkzeug Rose der Rational Software Cooperation angewandt; vgl. beispielsweise [Schu99], S. 181

[5] Dies ist natürlich nicht nur auf Throw-Aktionen beschränkt.

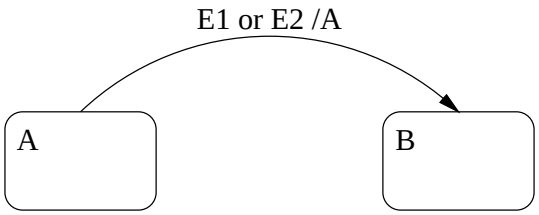


Abbildung 4.5: Transition mit mehreren Ereignissen

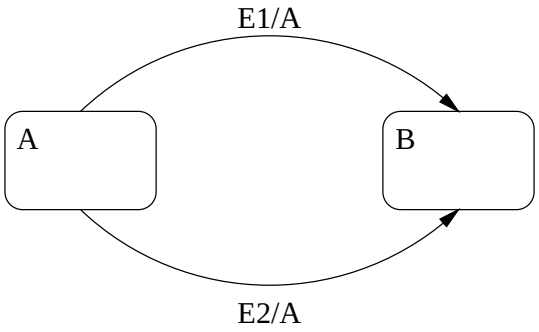


Abbildung 4.6: Analoge Transition mit nur einem Ereignis

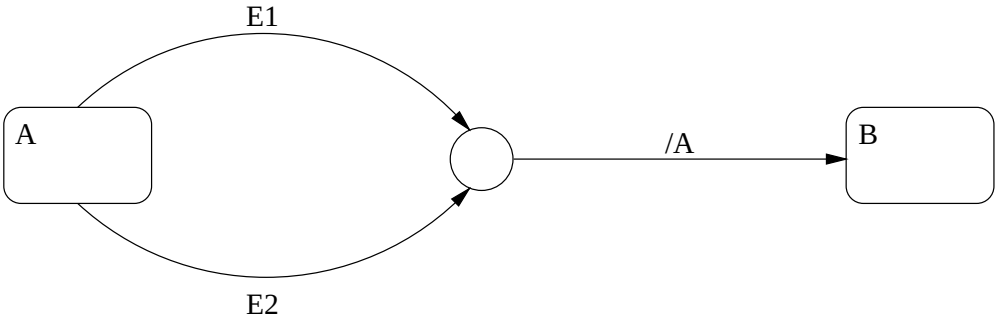


Abbildung 4.7: Analoge Transition mit Konnektor

Im Zusammenhang mit den Ereignissen muss noch erwähnt werden, dass in *SCed* einheitlich die schwache Rahmenbedingung[6] benutzt wird.

Die Aktionen

Aktionen dienen der Kommunikation der einzelnen Objekte untereinander sowie der Kommunikation nach außen. Wir implementieren die folgenden Aktionen, da wir davon ausgehen, hiermit ein recht allgemeines Spektrum an Möglichkeiten abgedeckt zu haben. Die Möglichkeit, weitere Aktionen nachzuimplementieren, ist aufgrund des objektorientierten Aufbaus der Klassenhierarchie für Aktionen leicht gegeben.

Folgende allgemeinen Aktionen (interne Aktionen) sind implementiert:

ThrowAktion Diese Aktion erzeugt ein Ereignis für den nächsten Zeitschritt. Sie wurde bereits oben erläutert.

SetVariable Hiermit können Variablen (siehe 4.2.4) gesetzt werden.

IncVariable Hiermit können Variablen um einen Wert, der auch negativ sein darf, erhöht werden.

JavaAktion Mit dieser Aktion wird beliebiger Java-Code ausgeführt. Diese Aktion wird nicht durch den Simulator interpretiert, sondern ist ausschließlich für den Quelltextexport bestimmt.

Während der Simulation wird stattdessen ein Popup-Fenster geöffnet, welches mitteilt, dass die entsprechende Java-Aktion jetzt ausgeführt wird.

ClearHistory/DeepClearHistory Mit diesen Aktionen kann man die History eines Zustands beziehungsweise aller darunterliegender Zustände löschen.

Weiterhin existieren die folgenden speziellen Aktion, um mit Dialogkomponenten kommunizieren zu können:

Dialog On/Off Mit diesen Aktionen werden Dialogkomponenten ein- bzw. ausgeschaltet. Dies hat je nach Dialogkomponente unterschiedliche Funktionen. Ein Timer wird damit zum Beispiel gestartet bzw. gestoppt, ein Radiobutton wird ein- bzw. ausgeschaltet.

SetDialogVisible Diese Aktion verbirgt oder zeigt Dialogkomponenten.

Aktionen können von den folgenden Komponenten erzeugt werden.

Dialogkomponenten werfen Aktionen in ihrer natürlichen Funktion. Das heißt, ein Button wirft eine Aktion, wenn er gedrückt wurde; ein Timer, wenn die Zeit abgelaufen ist, etc.

Zustände werfen je eine Aktion beim Betreten und beim Verlassen.

Übergänge werfen eine Aktion, wenn der Übergang ausgeführt wird.

[6] vgl. Abschnitt 2.5.2

Mit Aktion meinen wir in diesem Zusammenhang eine komplexe Aktion, also eine Menge der oben aufgezählten Basisaktionen. Diese Aktionsmenge kann gegebenenfalls auch die leere Menge sein. Es gibt keine Priorität bei der Ausführung der Aktionen. Theoretisch sollen sie gleichzeitig ausgeführt werden, was jedoch technisch nicht möglich ist. So werden sie in einer zufälligen Reihenfolge ausgeführt; es wird also keine Reihenfolge definiert. Sie werden jedoch alle innerhalb eines Zeitschrittes ausgeführt. Konflikte müssen in der Konstruktion vermieden werden.

Beispiel. Die zwei Aktionen *erhöhe Variable A um 2* und *setze Variable A auf 3*. Werden im gleichen Zeitschritt ausgeführt. *A* hatte bisher den Wert 0. Im nächsten Zeitschritt kann *A* den Wert 5 oder 3 haben![7]

Die Bedingungen

In unserer Beschreibung von Bedingungen wollen wir eine möglichst mächtige Ausdrucksweise implementieren. Wir beschreiben Bedingungen auch in unserer Implementation als aussagenlogische Ausdrücke, welche also Wahrheitswerte mit *und*, *oder* und *nicht* verknüpfen und die Auswertungsreihenfolge durch Klammerung beeinflussen.

Die aussagenlogischen Variablen, die in den Bedingungen verwendet werden können, stehen für folgende möglichen atomaren Bedingungen:

InState Die wohl häufigste Bedingung prüft, ob ein bestimmter Zustand aktiviert ist. So können Transitionen unter der Bedingung, dass das Statechart einen bestimmten Zustand hat, konstruiert werden.

HasHistory Diese Bedingung fragt einen Zustand, ob er eine History hat, also ob er bereits einmal betreten wurde und die History seitdem nicht mehr gelöscht wurde.

JavaBed Dieser Typ steht für eine frei in Java-Code formulierte Funktion, die einen Wert vom Typ *boolean* zurückliefert. Sie wird analog zur *JavaAktion* nicht in der Simulation geprüft, sondern nur für den Quelltextexport benutzt. Eine entsprechende Simulation muss hier mit Hilfe von Dialogkomponenten stattfinden.

VarBedingung Mit dieser Bedingung können Werte von Variablen überprüft werden. Diese Bedingung vergleicht eine Variable mit einem Wert, der eine Zahl, ein String oder eine andere Variable sein kann. Dabei sind die Operatoren $==$, $>=$, $<=$, $<$, $>$ und $<>$ möglich.

4.2.4 Die Variablen

Um den Umstand, dass Java-Bedingungen nicht im Simulator getestet werden können, ein wenig zu relativieren, haben wir Variablen eingeführt. Diese existieren während der Simulation in einem globalen Namensraum und können in Bedingungen mit den Operatoren *gleich* $==$, *ungleich* $<>$, *größer* $>$, *kleiner* $<$, *kleiner oder gleich* $<=$ und *größer oder gleich* $>=$ getestet und in Aktionen mit Zeichenketten oder nummerischen Werten gesetzt werden.

[7] Für eine genauere Beschreibung eines solchen Races sei auf Abschnitt 87 auf Seite 76 verwiesen.

Mit Hilfe dieser Variablen implementieren wir auch die Möglichkeit, die Harel[8] als parametrisierte Zustände vorgeschlagen hat:

> „In vielen Fällen [...] haben verschiedene Zustände identische interne Strukturen. Einige der häufigsten Situationen sind solche, die am besten als einziger Zustand mit einem Parameter angesehen werden."

Zur genaueren Betrachtung siehe Abschnitt 2.2.3 auf Seite 39.

Die Variablen können durch Aktionen gesetzt oder inkrementiert werden und können Zahlen oder Text enthalten. Der Typ wird dabei flexibel gehalten, das heißt, er ändert sich je nach Benutzung der Variable und wird entsprechend konvertiert.

Variablen entstehen während der Laufzeit dynamisch bei der ersten Benutzung. Wird eine Variable vor dem ersten Setzen in einer Bedingung getestet oder durch eine Aktion erhöht, so wird eine entsprechende Warnung ausgegeben.

Es gibt weiterhin zu jeder Dialogkomponente genau eine spezielle Variable, mit der diese beeinflusst oder überprüft werden kann.[9]

4.2.5 Die Zeit

In der Literatur gibt es zwei verschiedene Modelle die Zeit zu betrachten: diskret oder kontinuierlich.[10] Während sich das kontinuierliche Zeitmodell eher für theoretische Überlegungen eignet, ist das diskrete konfliktfreier in der Implementierung. Wir wählen dieses Modell, das auch für die Implementierung in Statemate gewählt wurde, da hier der Ablauf nicht von der Auswertungsreihenfolge abhängt und es prinzipiell immer problematisch ist, ein kontinuierliches Modell mit einer diskret arbeitenden Maschine zu simulieren.

Diskret bedeutet nun genau, dass während der Simulation verschiedene Phasen[11] unterschieden werden. Nur in bestimmten Phasen werden Aktionen gesammelt und anschließend „gleichzeitig" ausgewertet. Die dabei erzeugten Ereignisse sind dann erst im nächsten Schritt sichtbar, so dass sich die Ereignismenge nicht während der Bearbeitung ändern kann.

4.3 Die Dialoge

Damit der Benutzer mit dem Statechart während der Simulation interagieren kann, sind Dialoge notwendig. Da der Schwerpunkt dieser Arbeit jedoch nicht auf der computerunterstützten Erstellung von Dialogen liegt, beschränken wir uns auf ein recht rudimentäres Modell.

Wir haben genau einen Dialog, der in einen Benutzer- und einen Systemteil unterteilt ist. Zu Beginn der Simulation beziehungsweise des exportierten Programms wird er erzeugt und am Ende wieder gelöscht. Der Unterschied zwischen den beiden Teilen des Dialogs besteht im Quelltextexport, wo für den Benutzerteil der entsprechende Quellcode erzeugt wird, für den Systemteil jedoch nicht, da dieser zur Simulation von

[8] vgl. [Har87b] S. 258 f.
[9] vgl. auch Abschnitt 76 auf Seite 69 und Abschnitt 4.3
[10] siehe auch Abschnitt 2.8 auf Seite 54
[11] siehe Abschnitt 6.5.2 auf Seite 145

weiteren Ereignissen, wie dem Entnehmen einer Diskette etc. dient, die in der Implementation nicht durch einen Dialog realisiert werden.

Weiterhin werden die Komponenten in den Dialogen fortlaufend angeordnet, ohne die Möglichkeit, das Layout zu beeinflussen.

Jeder Dialogkomponente ist genau eine Variable zugeordnet, mit deren Hilfe diese Komponente beeinflusst werden kann.

Die Dialogkomponenten

Als Dialogkomponenten sind folgende Elemente vorgesehen, wobei eine leichte Erweiterbarkeit aufgrund des objektorientierten Aufbaus gegeben ist:

Button/Checkboxes/Radiobuttons Zum Auslösen von Aktionen, jedoch mit unterschiedlicher – vom Benutzer im Statechart zu modellierender – semantischer Bedeutung. Jedem Element ist genau eine Variable zugeordnet, mit der die Beschriftung der Komponente ausgelesen werden oder gesetzt werden kann.

Textfelder Lösen Aktionen aus, wenn sie geändert werden, und können mit einer zugehörigen Variable angesprochen werden, die den Inhalt des Textfeldes enthält.

Timer Sind nicht sichtbare Dialogkomponenten. Diese können im Dialog-Editor erzeugt und über die entsprechenden Aktionen angesprochen werden. Während der Simulation sind sie als Buttons sichtbar. Im Quelltextexport werden sie nicht als Dialogkomponenten exportiert. Auch hier ist eine Variable zugeordnet, die den Timerwert enthält.

Popups Sind Fenster, die eine Meldung enthalten. Hier kann der Text mit Hilfe der zugeordneten Variable gesetzt werden.

Weiterhin kann mit Hilfe der Aktion *setVisible* die Sichtbarkeit der Aktion geändert werden. Dabei wird bei der Komponente *Popup* das Fenster sichtbar oder unsichtbar, während die anderen Komponenten *enabled* oder *disabled* werden. Auf den Timer hat diese Aktion keinen Einfluss.

Wer fragt, ist ein Narr
für fünf Minuten.
Wer nicht fragt, bleibt es
ein für allemal.
(Deutsches Sprichwort)

Kapitel 5

Benutzerhandbuch

5.1 Installation

Um das Programm auf einem UNIX-System zu installieren, führen sie das Script `install <directory>` aus, wobei `<directory>` für das Zielverzeichnis steht. Sie benötigen dazu Schreibrechte für das Zielverzeichnis. Das Installationsscript benötigt die *bash-Shell*.

Auf Ihrem System muß ein lauffähiges Java 2 vorhanden sein. Um die exportierten Programme zu übersetzen benötigen sie auch ein JDK (mindestens Version 1.2.2) sowie das Programm `make`.

Unter anderen Systemen (wie zum Beispiel MacOS oder Windows) müssen Sie noch einige Anpassungen vornehmen und die Dateien manuell kopieren.

5.2 Überblick über das Programm *SCed*

Mit dem Programm *SCed* lassen sich Statecharts konstruieren, testen, simulieren und als Quelltext exportieren. Außerdem sind die in Abschnitt 69 auf Seite 69 beschriebenen Erweiterungen um Dialoge und frei formulierbaren Quelltext implementiert. Hierfür bietet *SCed* verschiedene Werkzeuge, die in diesem Kapitel genauer beschrieben werden.

Bevor wir die einzelnen Programmmodule beschreiben, hier noch ein paar Hinweise zur grundsätzlichen Bedienung des Programms:

- Das Programm ist für die Benutzung mit einer Maus konzipiert, dennoch gibt es für einige Menüfunktionen Tastaturbefehle, die dem erfahrenen Benutzer eine schnellere Bedienung ermöglichen.

- Zum Selektieren und Bearbeiten von Objekten wird ausschließlich die linke Maustaste verwendet. Mit der rechten Maustaste kann man bei einigen Objekten ein Kontextmenü erzeugen. Die mittlere Maustaste ist nicht belegt.

- Es kann immer nur ein Statechart gleichzeitig bearbeitet werden. Dieses kann jedoch in mehreren Ansichten und mit mehreren Werkzeugen geschehen. Jede Änderung in einer Ansicht kann automatisch auch alle anderen Ansichten beeinflussen, deshalb können einige Editiervorgänge das System für kurze Zeit verzögern.

- Einige Editiervorgänge werden bereits während der Eingabe auf Gültigkeit überprüft und gegebenenfalls zurückgewiesen. Wenn ein Editiervorgang keine offensichtliche Wirkung zeigt, prüfen Sie bitte, ob die Operation im Sinne der Statecharts erlaubt ist.

5.3 Der Inspektor

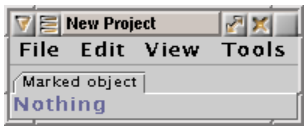


Abbildung 5.1: Inspektor bei Start des Programms

Beim Start des Programms öffnet sich automatisch der Inspektor (Abbildung 5.1) mit der Menüleiste sowie eine grafische Ansicht (Abbildung 5.2). Der Dialog-Editor, die Simulation sowie weitere Werkzeuge können bei Bedarf geöffnet werden.

Der Inspektor ist in gewisser Hinsicht das Hauptfenster des Programms. Er enthält die Menüleiste, aus der sämtliche weiteren Programmmodule gestartet werden können. Neben dieser enthält der Inspektor ein Informationsfeld, in dem die Eigenschaften der markierten Objekte angezeigt werden. Manche Eigenschaften können auch von hier direkt editiert werden. Einige Objekte werden durch Buttons dargestellt. Durch Drücken dieses Buttons wird dieses Objekt als markiertes Objekt ausgewählt.

Im folgenden möchten wir kurz die Möglichkeiten vorstellen, die der Inspektor für die verschiedenen Objekte bietet. Die Beispiele zeigen in der Regel ein maximales Inspektorfeld, also ein Feld in dem möglichste viele verschiedene Felder angezeigt werden. Felder, die nicht belegt sind, werden nicht angezeigt. So gibt es beispielsweise für einen atomaren Zustand kein Feld, welches die Unterzustände anzeigt.

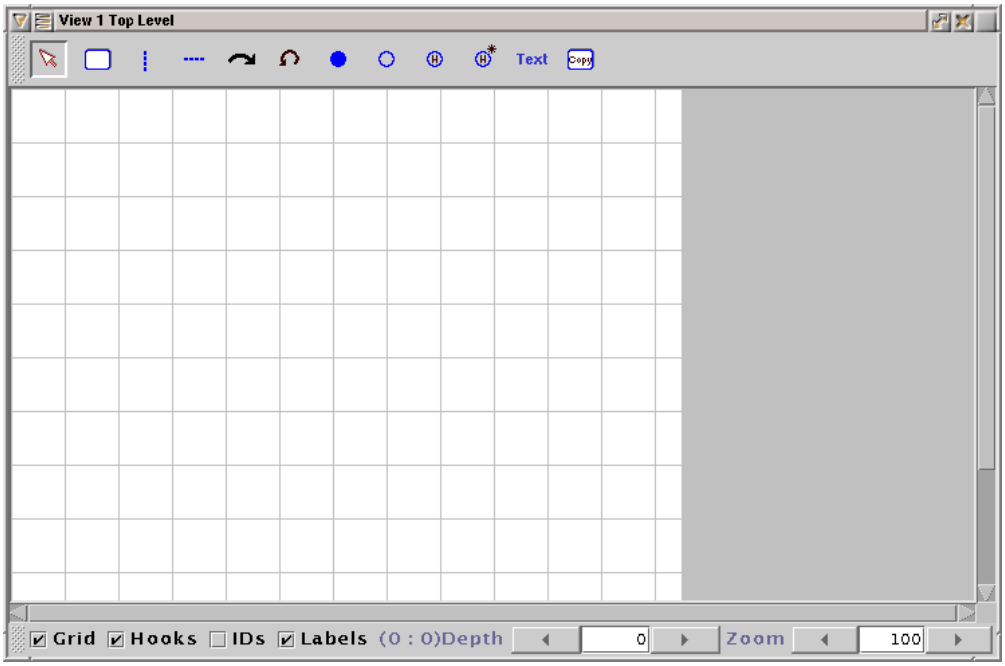


Abbildung 5.2: Grafische Ansicht

5.3.1 Inspektor für einen Zustand

Wenn ein Zustand markiert ist, zeigt der Inspektor ein Feld wie in Abbildung 5.3.

Dieses Beispiel zeigt einen Inspektor mit markiertem AND-Zustand. Im einzelnen sind hier angezeigt:

Typ Der Typ des Zustands (OR, ANDoder Atomar).

ID Eine eindeutige Nummer.

Name Ein frei zu vergebender Name. Dieses Feld kann vom Benutzer hier editiert werden.

Super Der eindeutige Oberzustand.

Incoming Eine Liste aller eingehenden Pfeile.

InActions Die Aktion, die ausgeführt wird, wenn der Zustand aktiviert wird.

Outgoing Eine Liste aller ausgehenden Pfeile.

OutActions Die Aktion, die ausgeführt wird, wenn der Zustand deaktiviert wird.

Sub Eine Liste aller Unterzustände.

Partitions Eine Liste der enthaltenen Partitionen.

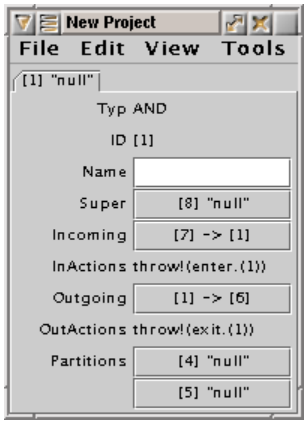


Abbildung 5.3: Inspektor mit markiertem AND-Zustand

Unterschiede bei einem atomaren Zustand

Bei einem atomaren Zustand gibt es zusätzlich eine Checkbox mit dem Namen *End-State*. Durch Selektieren dieser wird der Zustand zu einem Endzustand erklärt.

Unterschiede bei einem OR Zustand

Hier finden sich der Eintrag Sub, durch die einzelne Unterzustände markiert werden können.

5.3.2 Inspektor für einen Pfeil

Wenn ein Pfeil markiert ist, zeigt der Inspektor ein Feld wie in Abbildung 5.4.

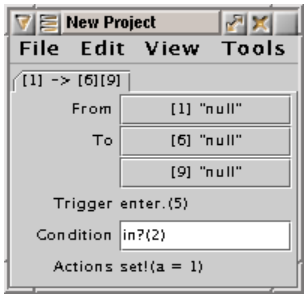


Abbildung 5.4: Inspektor mit markiertem Pfeil

Im einzelnen sind hier angegeben:

From Die Startknoten des Pfeils.

To Die Zielknoten des Pfeils.

Trigger Das Ereignis für den Übergang.

Condition Die Bedingung für den Übergang. Diese wird direkt in diesem Feld editiert.[1]

Actions Die Aktion, die bei einem Übergang ausgeführt wird.

5.3.3 Inspektor für einen Text

SCed bietet die Möglichkeit, Texte in das Statechart einzufügen. Diese haben allerdings keine semantische Funktion, sondern dienen dem Benutzer lediglich dazu, Anmerkungen zu einzelnen Elementen in die Ansichten zu integrieren.

Ein solcher Text kann als Bezugspunkt entweder den Zustand haben, in dem er liegt, oder einen Pfeil. Im Inspektor wird das entsprechende Objekt im Feld *Reference* angegeben; siehe hierzu auch Abbildung 5.5. Texte, die sich auf einen Pfeil beziehen, werden in der Form `Ereignis (Bedingung) / Aktion` mit den momentanen Werten des Pfeils initialisiert. Diese Voreinstellung kann im Inspektor geändert werden. Durch Drücken des Knopfes *ajust* werden wieder die aktuellen Werte des Pfeils übernommen.

Abbildung 5.5: Inspektor mit markiertem Textlabel

5.3.4 Weitere Elemente im Inspektor

Die anderen grafischen Elemente im Inspektor (Konnektoren, History, etc.) zeigen die kanonischen Eigenschaften des Objekts im Inspektor. Dies sind Kombinationen der oben beschriebenen.

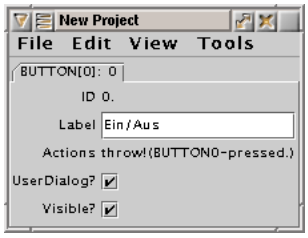


Abbildung 5.6: Inspektor mit markiertem Button

Dialogkomponenten werden im Inspektor wie in Abbildung 5.6 angezeigt. Hierbei sind

ID ist die eindeutige Dialogkomponenten ID.

[1] Für die Sprache der Bedingungen siehe Abschnitt 5.6.

Label ist ein frei zu vergebener Name.

Actions die von dieser Komponente ausgelösten Aktionen.

UserDialog? gibt an, ob diese Komponente auch im Export[2] berücksichtigt wird.

Visible? gibt an, ob die Komponente zu Beginn einer Simulation oder beim Start des exportierten Programms sichtbar ist.

5.3.5 Menüs

SCed verwendet zwei Arten von Menüs: einerseits gibt es die *Pull-down*-Menüs in der Menüleiste, die sich im Hauptfenster befinden und auf das gesamte Statechart beziehen, andererseits gibt es zu allen markierbaren Objekten (in der Zeichenfläche und im Dialog-Editor) ein *Popup*-Menü, das mit der rechten Maustaste geöffnet werden kann. Die Popup-Menüs für die einzelnen Elemente werden in Abschnitt 5.4.1 beschrieben.

Die Menüzeile des Hauptfensters besteht aus den folgenden Menüs:

File für das Laden und Speichern des Statecharts; außerdem können aus diesem Menü Ansichten gedruckt und das Programm verlassen werden

Edit für Undo/Redo und Makros

View zur Verwaltung der einzelnen Ansichten, der Ereignis- und Aktions-Listen und des Dialog-Editors

Tools mit den Werkzeugen Verifikator, Simulator und Quelltextexport

Einige der Funktionen aus diesen Menüs sind auch über spezielle Tastenkombinationen zu erreichen. Diese sind hinter den Menüeinträgen angegeben.

Das File-Menü

Das File-Menü besteht aus den folgenden Funktionen:

New (Alt-N) beginnt ein neues Statechart. Das bisherige Statechart und die erzeugten Dialogkomponenten werden gelöscht.

Open (Alt-O) Lädt ein neues Statechart über ein Auswahlfenster. Das bisherige Statechart und die erzeugten Dialogkomponenten werden gelöscht.

Save (Alt-S) Speichert das Statechart und den Dialog. Wenn das Projekt bereits einen Namen besitzt, wird dieser benutzt, ansonsten wird der Benutzer um eine entsprechende Eingabe gebeten.

Save As (Alt+Shift-S) Speichert das Statechart und den Dialog; der Benutzer wird hierbei zunächst nach einem Namen für das Projekt gefragt.

Print Bietet ein Untermenü, in dem alle momentan offenen Sichten aufgeführt sind. Wird eine dieser Sichten ausgewählt, so wird der darin angezeigte Zustand in das File `output.1` geschrieben. Diese Ausgabe findet im METAPOST-Format statt.

[2] siehe hierzu Abschnitt 5.11

Quit (Alt-Q) Beendet das Programm.

Das Edit-Menü

Das Edit-Menü bietet die folgenden Aktionen:

Undo (Alt-U) Nimmt die letzten Änderungen im Statechart oder im Dialog zurück. Wenn diese Funktion nicht ausgewählt werden kann, ist dieser Menüeintrag deaktiviert.

Redo (Alt+Shift-U) Stellt die letzten zurückgenommenen Änderungen im Statechart oder im Dialog wieder her. Änderungen, die nach dem letzten Undo vorgenommen wurden, werden dabei allerdings überschrieben, können aber mittels Undo wieder hergestellt werden. Wenn diese Funktion nicht ausgewählt werden kann, ist dieser Menüeintrag deaktiviert.

Align Size Mit dieser Funktion werden alle momentan markierten Konnektoren auf die gleiche Größe gebracht. *SCed* orientiert sich dabei am kleinsten der markierten Konnektoren. Sind weitere Elemente markiert, bleiben diese unberührt.

Insert Radiobuttons Mit diesem Menüeintrag wird ein Makro angesteuert, das eine vom Benutzer wählbare Anzahl von Radiobuttons sowohl in den Dialog als auch in das Statechart einfügt. Ereignisse und Aktionen werden entsprechend erzeugt und gesetzt.

Insert Checkboxes Mit diesem Menüeintrag wird ein Makro angesteuert, das eine vom Benutzer wählbare Anzahl von Checkboxes sowohl in den Dialog als auch in das Statechart einfügt. Ereignisse und Aktionen werden entsprechend erzeugt und gesetzt.

Das View-Menü

Über das View-Menü ist der Zugriff auf die folgenden Funktionen möglich:

Refresh (Alt-F) Aktualisiert sämtliche Ansichten und den Dialog-Editor.

Dialog Editor (Alt-D) Öffnet einen Dialog-Editor.[3] Es kann immer nur ein Dialog-Editor gleichzeitig offen sein.

Event List (Alt-E) Öffnet eine Liste aller Ereignisse. Hier können Ereignisse hinzugefügt, modifiziert oder gelöscht werden.

Action List (Alt-A) Öffnet eine Liste aller Aktionen. Hier können Aktionen hinzugefügt, modifiziert oder gelöscht werden.

New View (Alt-V) Öffnet eine neue Ansicht auf das globale Statechart.

New View of Marked State Wenn Zustände markiert sind, werden über diese Funktion Sichten auf diese Zustände geöffnet.

Das View-Menü endet mit einer Liste aller offenen Ansichten; durch Auswählen eines dieser Einträge wird die entsprechende Sicht in den Vordergrund geholt.

[3] Für eine genauere Beschreibung dieses Editors sei auf Abschnitt 5.5 verwiesen.

Das Tools-Menü

Im Tools-Menü lassen sich die zusätzlichen Werkzeuge auswählen.

Check (Alt-C) Startet den Verifikator. Die Ausgabe erscheint in einem separaten Fenster. Die möglichen Fehlermeldungen werden in Abschnitt 5.13 beschrieben.

Check Options Mit diesem Menüeintrag kann der Benutzer auswählen, welche der Eigenschaften des Statecharts überprüft werden sollen.

Run Simulation (Alt-R) Hiermit wird eine Simulation gestartet. Sollte das Statechart Fehler enthalten, wird zunächst eine entsprechende Warnung ausgegeben. Für die Funktionsweise des Simulators siehe Abschnitt 5.9.

Export Source (Alt-X) Wenn dieser Menüpunkt ausgewählt wird, wird das Statechart als ausführbarer Programmtext gespeichert, vergleiche hierzu auch Abschnitt 5.11.

Command Line (Alt-L) Öffnet ein Kommando-Fenster. Hier kann der Benutzer einzelne Befehle eingeben, die der in Anhang C angegebenen Grammatik entsprechen. Da hierbei jedoch direkt in die Struktur der Statechart-Modelle eingegriffen wird, ist mit diesem Werkzeug besondere Vorsicht geboten.

5.4 Der grafische Editor

Der grafische Editor (Abbildung 5.2) dient dem Editieren eines Statecharts. Er ist unterteilt in *Werkzeugleiste*, *Zeichenfläche* und *Eigenschaftsleiste*, welche wir im folgenden beschreiben. Die Eigenschaftsleiste und die Werkzeugleiste können innerhalb der grafischen Ansicht an allen Rändern positioniert werden. Sie können auch vom Fenster abgekoppelt in einem eigenen Fenster dargestellt werden.

5.4.1 Zeichenfläche

Bevor wir das Zeichenfenster erklären, wollen wir hier erst kurz aufzählen, wie Objekte dargestellt werden.

Zustände

Der Kern der Statecharts sind wohl die Zustände, die wir in drei Typen in unserem Programm unterstützen: atomare, AND- und OR-Zustände. Sie werden in der Zeichenfläche als Rechtecke mit abgerundeten Kanten dargestellt. Atomare Zustände haben einen schwarzen Rand, AND- und OR-Zustände einen blauen. So können komplexe von atomaren Zuständen auch dann unterschieden werden, wenn die logische Darstellungstiefe die Unterzustände nicht anzeigen würde. AND-Zustände unterscheiden sich von OR-Zuständen dadurch, dass sie eine Unterteilung (Partitionierung) haben. Diese wird durch eine horizontale oder vertikale gestrichelte Linie dargestellt. Wird ein atomarer Zustand als Endzustand markiert, so wird er invertiert dargestellt (nur atomare Zustände können Endzustände sein). Abbildung 5.7 zeigt ein paar Beispielzustände.

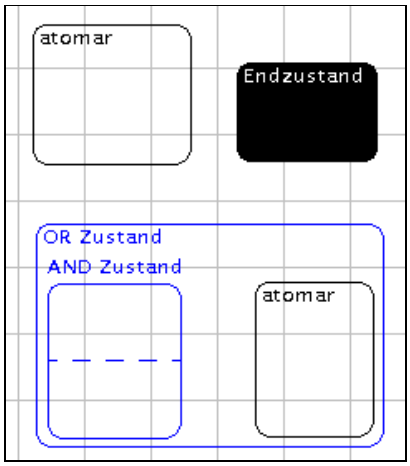


Abbildung 5.7: Verschiedene Zustände in der Ansicht

Das Popup-Menü für Zustände Das Popup-Menü, das für jeden Zustand aufgerufen werden kann, enthält die folgenden Einträge:

State-Properties Öffnet einen neuen Inspektor, der sich auf den Zustand bezieht.

View State Öffnet eine Ansicht für diesen Zustand.

Delete State Löscht diesen Zustand. Hierbei ist zu beachten, dass Unterzustände nicht gelöscht werden, sondern lediglich in der Zustandshierarchie eine Ebene aufrücken.

State In Actions Bietet die Möglichkeit, die Aktionen, die beim Betreten des Zustands ausgelöst werden, zu modifizieren.

State Out Actions Hierüber können die Aktionen, die beim Verlassen des Zustands ausgelöst werden, verändert werden.

Das Popup-Menü für Partitionsgrenzen Bei Partitionsgrenzen besteht das Menü nur aus einem einzigen Eintrag:

Delete Partitions Hiermit werden sämtliche Partitionsgrenzen eines AND-Zustands gelöscht. Der Zustand wird dadurch wieder zu einem OR-Zustand.

Transitionen

Transitionen zwischen verschiedenen Zuständen werden durch Kantenzüge von Strecken dargestellt. An jedem Schnittpunkt zwischen zwei Strecken und dem Anfangs- und Endpunkt wird ein Anfasshaken oder *Hook* (ein kleines Quadrat) dargestellt. Mit diesen kann die Position der Verbindungspunkte bestimmt werden. Der Kantenzug hat eine Richtung die durch eine Pfeilspitze dargestellt wird. Es können beliebig viele Hooks in eine Transition eingefügt werden, so dass deren Verlauf frei modelliert werden kann. Die Hooks dienen auch dazu, die Kante zu markieren. Abbildung 5.8 zeigt einige Beispiele.

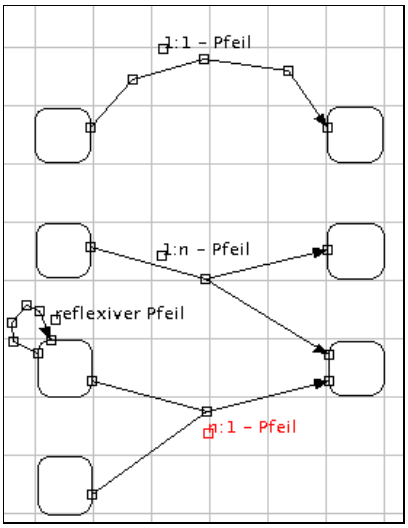


Abbildung 5.8: Verschiedene Transitionen in der Ansicht

Das Popup-Menü für Übergänge Genaugenommen können Pfeile nicht direkt angeklickt werden; deshalb gibt es auch kein Popup-Menü für sie. Stattdessen gibt es ein Popup-Menü für die Hooks, das die folgenden Einträge enthält:

Arrow-Properties Öffnet einen neuen Inspektor, der sich nur auf diesen Pfeil bezieht.

Label Arrow Fügt zu dem Hook einen Text hinzu, der sich auf den zugehörigen Pfeil bezieht. Die Position dieses Textes bezieht sich immer auf den Hook, zu dem er erzeugt wurde.[4]

Double Hook Mit dieser Funktion werden zu einem Pfeil neue Hooks hinzugefügt. Dies ist nicht bei Hooks möglich, bei denen mehrere Abschnitte beginnen oder enden; in diesem Fall ist der Menüeintrag deaktiviert.

Arrow Actions Bietet die Möglichkeit, die Aktionen des Übergangs zu modifizieren.

Trigger Event Hierüber kann das Ereignis ausgewählt werden, das den Übergang auslösen soll.

Delete Hook Diese Funktion löscht einen einzelnen Hook. Das ist nicht möglich, wenn es sich um einen Hook handelt, der am Anfang oder Ende eines Pfeils liegt, oder wenn mehrere Abschnitte bei ihm beginnen oder enden; in diesen Fällen ist dieser Menüeintrag deaktiviert.

Delete Section Hierüber kann ein Abschnitt eines Pfeils entfernt werden. Dies ist nicht möglich, wenn mehrere Abschnitte bei dem ausgewählten Hook beginnen oder enden; in diesem Fall ist dieser Menüeintrag deaktiviert.

Delete Arrow Mit diesem Menüeintrag kann ein Pfeil komplett gelöscht werden.

[4] vgl. Abschnitt 5.3.3

Konnektoren

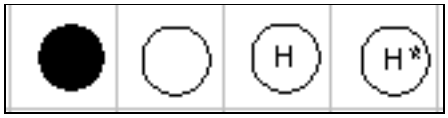


Abbildung 5.9: Defaulteinstieg, Konnektor, Historyeinstieg und tiefer Historyeinstieg in der Ansicht

Unter dem Begriff Konnektoren fassen wir hier die Standard-Konnektoren, Defaulteinstiege, History- und tiefe Historyeinstiege zusammen. Sie alle werden durch einen Kreis dargestellt. Nur bei den Defaulteinstiegen ist dieser gefüllt. Historyeinstiege enthalten ein H und tiefe Historyeinstiege enthalten H*. Beispiele zeigt Abbildung 5.9.

Das Popup-Menü für Konnektoren Bei Konnektoren enthält das Popup-Menü nur die folgenden zwei Einträge:

Connector-Properties Öffnet einen Inspektor, der sich nur auf diesen Konnektor bezieht.

Delete Connector Löscht diesen Konnektor.

Textfelder

Als Erweiterung ist es möglich, das Statechart mit Textfeldern zu beschriften. Die Eigenschaften dieser Textfelder werden im Inspektor angezeigt (siehe Abschnitt 5.3.3). Das Popup-Menü enthält nur den einen Eintrag *Delete Text*.

5.4.2 Werkzeugleiste

Abbildung 5.10: Die Werkzeugleiste

In der Werkzeugleiste (Abbildung 5.10) wählt man das Werkzeug aus, mit dem das Statechart bearbeitet werden soll. Im einzelnen sind dies die folgenden:

Mit dem Pfeil können Objekte markiert werden. Hierzu klickt man einmal kurz in das zu markierende Objekt. Wird dabei die *Strg*-Taste gedrückt gehalten, so wird das Objekt zusätzlich zu den vorhandenen markiert. Markierte Objekte werden rot dargestellt.

Durch Drücken und Schieben können die Objekte an eine neue Position bewegt werden. Achten Sie hierbei auch auf das Feedback. Bei einer ungültigen Position wird das Objekt nicht dargestellt. Wenn Sie Zustände in oder aus anderen Zuständen herausschieben, ändert sich gegebenenfalls der Typ des darüberliegenden Zustands von atomar auf OR oder umgekehrt!

Einige Objekte (Zustände und Konnektoren) können in ihrer Größe skaliert werden. Bewegen Sie den Pfeil dazu in die rechte untere Ecke des Objekts, bis der Cursor zu einem Pfeil nach rechts unten wird. Dann können Sie durch Drücken und Schieben die neue Größe einstellen. Bei Zuständen ist die neue Größe allerdings nur dann gültig, wenn durch die Veränderung keine Unterzustände hinzukommen oder aus dem Zustand hinausgeraten.

Die folgenden Objekte können verschoben werden: Zustände, Hooks, alle Konnektoren (Default, History, etc.), Partitionsgrenzen und Textfelder. Partitionsgrenzen können jedoch nur so verschoben werden, dass sich die Zuordnung der Unterzustände zu den Partitionen dadurch nicht ändert.[5]

Mit diesem Werkzeug können Sie einen neuen Zustand einfügen. Bewegen Sie den Pfeil an die Stelle, an der der Zustand eingefügt werden soll. Drücken Sie nun die linke Maustaste und bestimmen Sie die Größe des neuen Zustands. Zum Einfügen die Maustaste loslassen. An einer ungültigen Position (z. B. wenn der Zustandsrand einen anderen Zustand schneiden würde) kann kein neuer Zustand eingefügt werden.

Wird der Zustand so positioniert, dass er andere Zustände enthält, so wird er als OR-Zustand erschaffen. Ansonsten als atomarer Zustand.

Mit diesen beiden Werkzeugen fügen Sie in Zuständen Partitionen ein. Bewegen Sie den Cursor in einen Zustand und sehen Sie, wie die Partition den Zustand teilen würde. Bestätigen Sie das Einfügen durch Drücken der linken Maustaste.

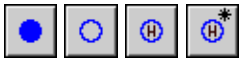
Mit diesem Werkzeug fügen Sie eine neue Verbindung in das Statechart ein. Bewegen Sie den Cursor über die Zeichenfläche. Bei einer gültigen Startposition wird der Mauspfeil zu einem Fadenkreuz. Drücken Sie an der gewünschten Startposition die linke Maustaste. Ziehen Sie den Cursor auf die gewünschte Endposition (auch hier wird der Cursor bei gültigen Positionen zu einem Fadenkreuz) und lassen Sie die linke Maustaste los. Die neue Kante wird eingefügt, wobei Start- und Endpunkte automatisch auf die nächstgelegenen Ränder der Objekte gesetzt werden.

Start und Endpunkte für Kanten können sein: Zustände, Hooks (mit Einschränkungen), Defaulteinstiege (nur Startpunkte), Konnektoren und Historykonnektoren.

[5] Hierbei beziehen sich die Positionen der Unterzustände grundsätzlich auf die linke obere Ecke der Partition, in der sie liegen. Diese relativen Positionen verändern sich durch das Schieben von Partitionsgrenzen nicht.

Mit diesem Werkzeug kann *keine* Kante von einem Zustand auf sich selbst gezogen werden. Der Versuch dieser Aktion bricht das Kantenziehen genauso ab, wie das Ziehen auf eine ungültige Position. Um eine Kante von einem Zustand auf sich selbst zu ziehen, benutzen Sie bitte das folgende Werkzeug.

Mit diesem Werkzeug können Sie eine reflexive Kante an einen Zustand setzen. Bewegen Sie den Cursor über den Zustand und drücken Sie die linke Maustaste. Eine Kante von diesem Zustand zu sich selbst mit zwei mittleren Hooks wird nun automatisch eingefügt.


Mit diesen Werkzeugen fügen Sie einen Defaulteinstieg, einen Konnektor, einen Historyeinstieg oder einen tiefen Historyeinstieg ein. Dazu bewegen Sie den Cursor an die Position, an der das Objekt eingefügt werden soll, drücken die linke Maustaste und ziehen es auf die gewünschte Größe. Bei ungültigen Positionen wird kein Feedback-Objekt angezeigt.

Mit diesem Werkzeug können Sie ein Textfeld einfügen. Bewegen Sie den Cursor an die gewünschte Position und drücken Sie die linke Maustaste.

Mit diesem Werkzeug können Sie einen Zustand inklusive seiner Unterzustände kopieren. Die Benutzung erfolgt genauso wie beim Verschieben von Zuständen, mit dem Unterschied, dass nach der Aktion Original *und* Kopie existieren. Die

Abbildung 5.11: Die Ersetzungstabelle

in den Zuständen und Kanten benutzten Ereignisse und Variablen können mit Hilfe der Ersetzungstabelle (siehe Abbildung 5.11) durch neue Ereignisse bzw. Variablen ersetzt werden. Die Ereignisse, die sich auf Zustände beziehen, die kopiert wurden, werden in der Übersetzungstabelle automatisch mit den neuen Zuständen angegeben. Die Namen der Zustände werden unverändert übernommen, so dass eventuelle Umbenennungen von Hand durchzuführen sind. Die

Eindeutigkeit von Abfragen, die sich auf die Zustände beziehen, wird durch die intern verwalteten Identifier sichergestellt.

5.4.3 Die Eigenschaftsleiste

Abbildung 5.12: Die Eigenschaftsleiste

Mit der Eigenschaftsleiste (Abbildung 5.12) der grafischen Ansicht können Sie einstellen, wie Objekte dargestellt werden sollen.

Grid Soll das graue Gitter in der Zeichenfläche dargestellt werden?

Hooks Sollen die Hooks (Anfasser) an den Pfeilen dargestellt werden? Achtung: Wenn diese nicht dargestellt werden, können die Pfeile nicht direkt mit der Maus bearbeitet werden.

IDs Sollen die ID's an den Zuständen angezeigt werden?

Labels Sollen die Textfelder dargestellt werden?

Depth Die logische Tiefe[6] der Darstellung.

Zoom Die optische Vergrößerung der Darstellung in Prozent.

5.5 Der Dialog-Editor

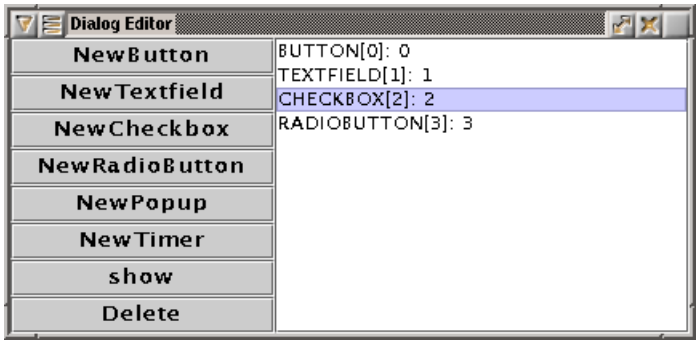


Abbildung 5.13: Der Dialog-Editor

Der Dialog-Editor (siehe Abbildung 5.13) bietet die Möglichkeit, die für die zu modellierende Anwendung benötigten Dialoge zu erstellen. Er ist recht einfach gehalten und setzt das Layout des Dialogs automatisch.

[6] Es ist hier die globale Tiefenvergrößerung implementiert. Siehe hierzu auch Abschnitt 66 ab Seite 59.

Mit Hilfe der Schaltflächen können entsprechende Dialogkomponenten hinzugefügt werden. Mit *show* wird der bisher erstellte Dialog angezeigt und mit *Delete* kann eine gewählte Dialogkomponente wieder entfernt werden.

Die Eigenschaften der Dialogkomponente werden mit Hilfe des Inspektors editiert. So kann für jede Dialogkomponente gesetzt werden, ob sie eine *UserDialog*-Komponente ist (voreingestellt) oder ob sie im *SystemDialog*[7] erscheinen soll. Weiterhin wird gesetzt, ob sie standardmäßig aktiviert oder deaktiviert und sichtbar oder unsichtbar sein soll. Außerdem gibt es bei jeder Dialogkomponente ein Popup-Menü, das über die rechte Maustaste erreichbar ist. Hierin können die Aktionen eingestellt werden, die von dieser Komponente ausgeführt werden.

5.5.1 Untypische Dialogkomponenten

Als untypische Dialogkomponenten bezeichnen wir Elemente, die nur im erweiterten Sinn zum Dialog gezählt werden können; dazu gehören Popups und Timer sowie Java-Ersatzkomponenten. Sie unterscheiden sich von den üblichen Dialogkomponenten dadurch, dass sie im Benutzerdialog des exportierten Statecharts nicht erscheinen. Die Bezeichnung als Dialogkomponenten liegt darin begründet, dass diese Elemente dem Statechart ebenfalls mit dem Dialog-Editor hinzugefügt werden können.

Popup

Diese Dialogkomponente wird natürlich nicht im User- oder Systemdialog angezeigt, sondern bekommt – wie für Popup-Fenster üblich – ein eigenes Fenster. Insofern ist dies eine untypische Dialogkomponente.

Timer

Der Timer ist auch keine Dialogkomponente im üblichen Sinne. Da er jedoch ein ähnliches Verhalten im Zusammenhang mit Statecharts zeigt (Er erzeugt zu einem bestimmten Zeitpunkt eine Aktion) wird er auch hier erstellt. Der Timer erzeugt seine Aktion dann, wenn er abgelaufen ist. Während der Simulation wird für den Timer zusätzlich ein Button erzeugt, so dass die Timer-Aktion auch manuell ausgelöst werden kann. Im Quelltextexport ist dieser nicht vorhanden.

Java-Ersatzkomponenten

Da in der Simulation keine Java-Aktionen ausgeführt oder Java-Bedingungen getestet werden können, werden Ersatzkomponenten erzeugt.

Für eine Java-Aktion wird ein Popup erzeugt, der angibt, das die entsprechende Java-Aktion jetzt ausgeführt wurde. Für eine Java-Bedingung wird eine Checkbox erzeugt, so dass man den Wahrheitswert der Bedingung im System-Dialog setzen kann.

Die Ersatzkomponenten werden alle als Komponenten für den Systemdialog erzeugt, so dass sie später im Export nicht mehr erscheinen.

[7] siehe hierzu auch Abschnitt 94 ab Seite 80

5.5.2 Kommunikation über Variablen

Um die Werte aus Dialogkomponenten auszulesen oder deren Werte zu setzen, gehört zu jeder Dialogkomponente genau eine Variable, die deren Inhalt repräsentiert. Der Name dieser Variablen bildet sich aus dem Prefix `DLG` und der ID der Dialogkomponente. So ist `DLG5` die Variable der Dialogkomponente mit der ID 5. Mit dieser Variablen können die folgenden Werte gelesen bzw. gesetzt werden:

Button, Checkbox, Radiobutton Hier wird in der Variablen der Name dieser Komponenten verwaltet.

Timer Hier wird der Timerstartwert verwaltet.

Textfield Der Inhalt des Textfeldes steht in der Variablen.

Popup Der Meldungstext steht in der Variablen.

5.6 Bedingungen

Bedingungen schränken mögliche Transitionen ein. Sie gehören also zu einer Transition und werden im Inspektor editiert. Grundsätzlich können hier aussagenlogische Formeln eingegeben werden, wobei folgende Punkte zu beachten sind:

- Die Ausdrücke werden mit geschweiften Klammern geklammert.
- Für die logische Konjunktion wird das Schlüsselwort `AND` oder `&&` benutzt.
- Für die logische Disjunktion wird das Schlüsselwort `OR` oder `||` benutzt.
- Die logische Negation wird durch `NOT` oder `!` beschrieben.
- Es gibt keine voreingestellte Auswertreihenfolge. Kombinationen der obigen logischen Operationen müssen deshalb geklammert werden.

Für die aussagenlogischen Variablen gibt es folgende Möglichkeiten:

IN STATE Mit dieser Bedingung wird gefragt, ob ein Zustand gerade aktiviert ist. Das Format ist `IN STATE <ID>` oder `in?(<ID>)`, wobei `<ID>` ein Platzhalter für die ID des Zustands ist. Wenn die angegebene ID noch nicht existiert, wird die Bedingung zurückgewiesen.

HAS HISTORY fragt, ob ein Zustand eine History hat. Das Format ist `HAS HISTORY <ID>` oder `hasHist?(<ID>)`. Diese Bedingung wird zurückgewiesen, wenn die angegebene ID nicht die ID eines OR-Zustands oder einer Partition ist. Die ID wird automatisch geändert, wenn der Zustand nachträglich zu einem AND-Zustand gemacht wird.

VARIABLE Mit dieser Bedingung können Variablen getestet werden. Das Format ist `VARIABLE <VARNAME> <OP> <VALUE>` oder `var?(<VARNAME> <OP> <VALUE>)`. Hierbei ist `<VARNAME>` der Name der Variablen, `<OP>` ist der Vergleichsoperator. Dieser darf einer der folgenden sein: `==`, `>=`, `<=`, `<`, `>`, `<>`. Als

Vergleichswert `<VALUE>` kann eine ganze Zahl, eine Fließkommazahl, eine Zeichenkette (in Anführungszeichen) oder eine andere Variable dienen (Zeichenkette ohne Anführungszeichen). Zu Beachten ist, dass Variablen, deren Namen mit dem Prefix `DLG` beginnt, Dialogvariablen sind und den Inhalt von Dialogkomponenten repräsentieren. Der Verifikator kann prüfen, ob Variablen getestet werden, ohne dass sie irgendwo gesetzt werden. Genauso kann er prüfen, ob Variablen gesetzt werden, ohne dass sie irgendwo geprüft werden. Dialogvariablen sind von dieser Prüfung ausgeschlossen.

JAVA Java-Bedingungen sind frei definierbare Java-Funktionen mit booleschem Rückgabewert in der Form `JAVA <FUNCTION>` oder `java?(<FUNCTION>)`. Diese werden in der Simulation nicht getestet, deshalb wird für jede Java-Bedingung automatisch eine Systemdialogkomponente vom Typ Checkbox erzeugt, mit welcher der Wahrheitswert dieser Funktion eingestellt werden kann. Im Quelltextexport wird dann die eingetragene Funktion exportiert. Der Editor erlaubt jedoch keine Zeilenumbrüche, so dass die Funktion in einer Zeile formuliert werden muss. Es empfiehlt sich, hier nur Platzhalterfunktionen einzutragen und die Funktionen später im exportierten Quelltext zu formulieren. Beispiel: `java?("boolean f1() // Hier soll f1 stehen")`.

5.7 Ereignisse

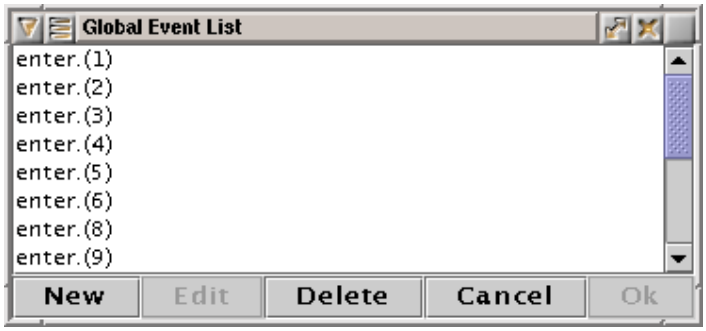


Abbildung 5.14: Die Ereignisliste

Die Ereignisse können über die Event-Liste (Abbildung 5.14) verwaltet werden. Ereignisse werden in *SCed* nur durch ihren Wert in der Zeichenkette unterschieden. Sie haben keine weiteren Eigenschaften. Es werden für die meisten Objekte beim Erschaffen gleich entsprechende Ereignisse mit entsprechenden Namen miterzeugt, welche man natürlich nach Belieben ändern kann.

Ereignisse dienen einer Transition als Trigger. Man stellt ein Ereignis als Auslöser über das Kontextmenü[8] der Transition ein. Wird kein Ereignis festgelegt, so gilt das Ereignis als immer ausgelöst.

[8] siehe Abschnitt 5.4.1

5.8 Aktionen

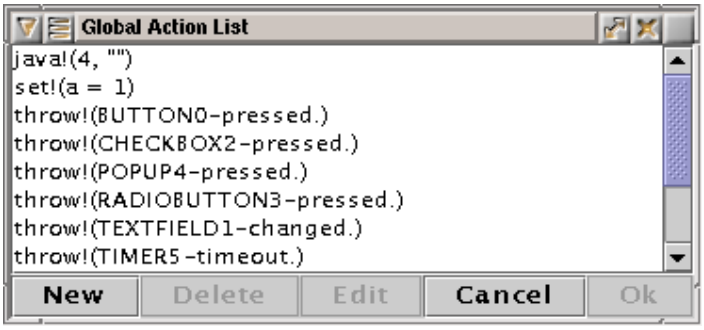


Abbildung 5.15: Die Aktionsliste

Fast alle Objekte können Aktionen auslösen. So löst zum Beispiel jede Dialogkomponente Aktionen aus; Zustände lösen beim Betreten und Verlassen Aktionen aus und Transitionen, wenn sie durchgeführt werden. Diese können frei spezifiziert werden. Einen Überblick über die bereits definierten Aktionen liefert die Aktionsliste (Abbildung 5.15). Die folgenden Aktionen sind implementiert:

Java Action Dies ist eine frei in Java programmierbare Aktion. Ihr Code wird in der Simulation nicht interpretiert, jedoch im exportierten Programm. In der Simulation wird ein Popup geöffnet, welches anzeigt, dass diese Aktion jetzt ausgeführt worden wäre.

Clear History Diese Aktion löscht im spezifizierten Zustand die History.

Deep Clear History Diese Aktion löscht im spezifizierten Zustand und allen darin liegenden Zuständen die History.

Throw Event Diese Aktion „wirft“ das spezifizierte Ereignis.

Inc Variable Diese Aktion erhöht die Variable um den angegebenen Wert. Auch negative Werte sind möglich. Bei Zeichenkettenvariablen, wird die Zeichenkette an die vorhandene Variable angehängt.

Set Variable Diese Aktion setzt eine Variable auf einen Wert. Der Typ der Variablen (String, Integer oder Double) kann beliebig wechseln und wird, wenn möglich, entsprechend konvertiert.

Set Dialog Element Visible Diese Aktion macht eine Dialogkomponente sichtbar oder unsichtbar. Der Timer ignoriert diese Aktion.

Dialog On und Off Diese Aktion schaltet eine Dialogkomponente an oder aus. Was dies bedeutet hängt von der Dialogkomponente ab:

Timer Beim Timer wird mit dieser Aktion der Timer gestartet bzw. gestoppt.

Checkboxes und Radiobuttons Diese Komponenten werden mit der On/Off-Aktion selektiert oder deselektiert.

alle anderen Komponenten ignorieren diese Aktion.

Sollte aus diesem Fenster eine Aktion zum Löschen ausgewählt werden, so wird geprüft ob diese Aktion von Komponenten noch benutzt wird. Wenn dies der Fall ist, wird eine entsprechende Warnung ausgegeben. Wird die Aktion trotzdem gelöscht, so wird sie automatisch auch aus allen Komponenten ausgetragen, die sie benutzen.

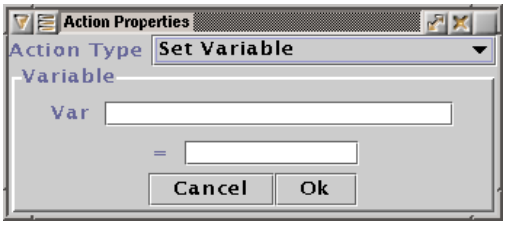


Abbildung 5.16: Das Aktionen-Eigenschaftsfenster

Wenn eine neue Aktion erzeugt oder eine vorhandene editiert werden soll, so wird das Aktions-Eigenschaftsfenster geöffnet (siehe Abbildung 5.16). Hier wählt man mit dem Selektor den Typ der Aktion aus[9] und setzt die Parameter neu. Welche Parameter das sind, hängt vom Typ der Aktion ab.

5.9 Der Simulator

Der Simulator wird aus dem Hauptmenü aufgerufen. Mit Hilfe des Simulators ist es möglich, das Verhalten eines Statecharts über der Zeit zu simulieren. Zu Beginn der Simulation wird das Statechart mit Hilfe des Verifikators (siehe Abschnitt 5.13 ab Seite 127) geprüft. Wenn es Fehler enthält, erscheint eine Warnung.

Als nächstes öffnet sich das Simulationsfenster (siehe Abbildung 5.17). Dieses enthält folgende Elemente:

Steuerknöpfe Mit diesen Knöpfen läßt sich die Simulation steuern. So wird diese mit *Run* gestartet und mit *Stop* wieder gestoppt. Mit *Step* kann die Simulation ein Zeitschritt weitergesetzt werden, *Phase* schaltet eine Phase[10] weiter. Mit *Reset* kann die Simulation wieder in den Startzustand gesetzt werden, und mit *Close* wird die Simulation beendet und das Simulationsfenster geschlossen.

Delay Mit diesem Wert wird die Zeit eingestellt, welche die Simulation in der Phase *Collect External Events* auf Benutzerevents wartet. Der Minimalwert ist hier 100 ms.

Logmeldungen erscheinen direkt im Fenster unter der Schalterleiste.

Variable In dieser Liste werden alle in der Simulation bisher benutzten Variablen und ihr Wert gelistet. Diese können bei angehaltener Simulation hier auch markiert und dann im Inspektor gesetzt werden.

[9] wenn diese neu erschaffen wird
[10] siehe hierzu auch Abschnitt 6.5.2

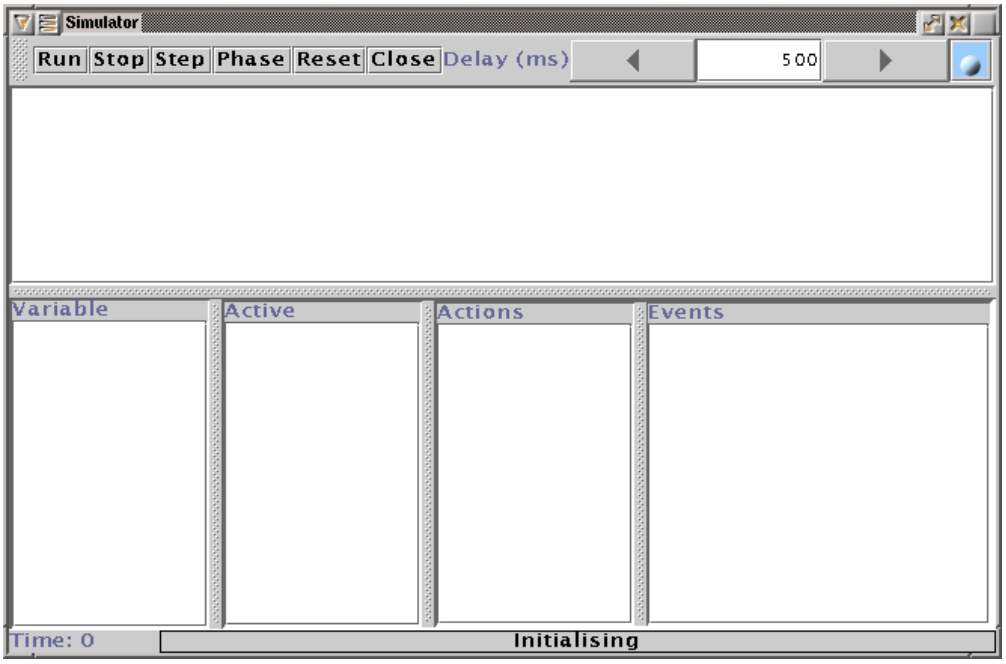


Abbildung 5.17: Das Simulationsfenster

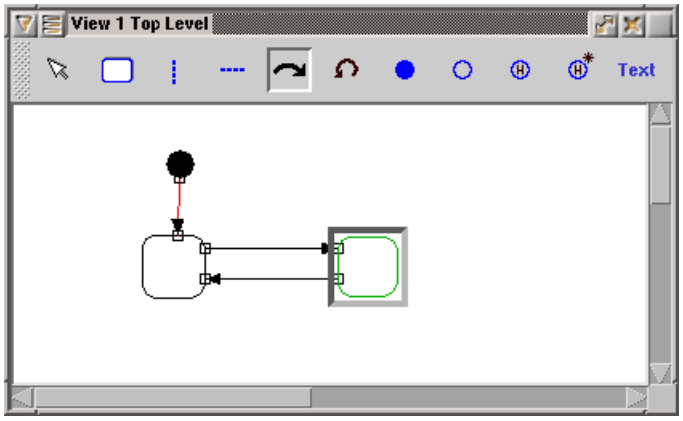


Abbildung 5.18: Ansicht während einer Simulation

Active ist die Liste aller momentan aktiven Zustände. Diese werden auch in allen Ansichten durch eine grüne Farbe und einen Rahmen hervorgehoben. Siehe Abbildung 5.18.

Actions sind alle in diesem Zeitschritt bisher erzeugten Aktionen.

Events sind alle in diesem Zeitschritt bisher geworfenen Ereignisse.

Time ist die Zeit in Schritten gemessen.

Status gibt die aktuelle Phase an.

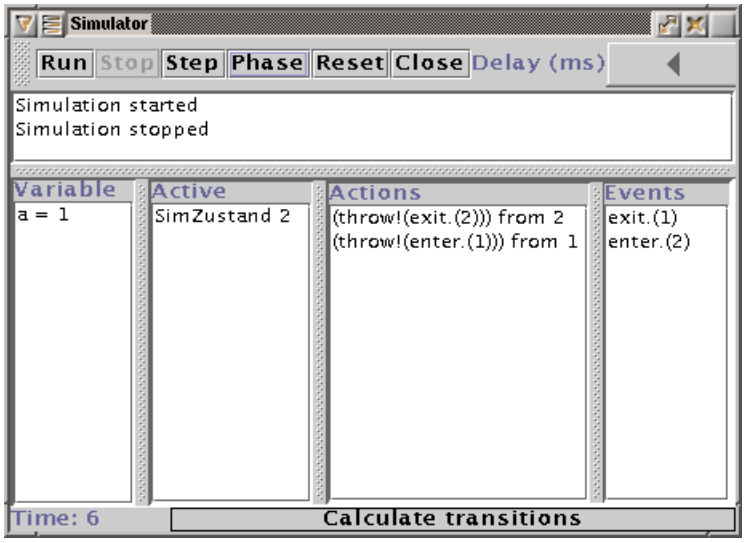


Abbildung 5.19: Simulationsfenster während einer Simulation

Während ein Simulationsfenster geöffnet ist, bieten die Editoren eine etwas andere Funktionalität. Durch selektieren eines Zustands mit dem Pfeil-Werkzeug, wird der entsprechende Zustand aktiviert bzw. deaktiviert. Die dabei generierten Aktionen werden ignoriert. Der Simulationsablauf kann jedoch nach einer solchen Manipulation fortgesetzt werden.

Weiterhin können bei angehaltener Simulation Variablen im Simulationsfenster markiert werden. Diese erscheinen dann im Inspektor und können dort mit einem neuen Wert gesetzt werden. Auch hiernach ist eine Fortsetzung der Simulation möglich.

5.10 Undo/Redo

Sollten Sie einen Editiervorgang rückgängig machen wollen, so wählen Sie einfach aus dem Menü die Funktion *Undo* aus. Fast alle Editiervorgänge (außer kleinen Änderungen im Inspektor) lassen sich so bis zu 50 Schritte rückgängig machen. Mit der Funktion *Redo* können Sie bis zu 50 Undo-Vorgänge wieder herstellen.

5.11 Quelltextexport

5.11.1 Exportieren des Statecharts

Mit dem Menüpunkt *Export source* kann das erstellte Statechart als Java-Quellcode exportiert werden. In dem folgenden Dialog wählt man das Zielverzeichnis aus. Dort werden die Dateien *Export.java*, *ExportAkt.java* und *ExportBed.java* erzeugt. Diese Dateien können mit dem *Makefile* unterhalb von *export* übersetzt (`make build`) und ausgeführt (`make run`) werden.

5.11.2 Methoden der Quelltextsegmente

In den Quelltextsegmenten, die in Java-Aktionen und -Bedingungen eingegeben werden können, sind neben sämtlichen Klassen und Methoden, die von Java generell zur Verfügung gestellt werden, die in den Tabellen dieses Abschnitts aufgeführten Konstruktionen möglich, die auf das Statechart zugreifen. Verwendete Klassen müssen nach dem Export in den Klassen *ExportAkt* beziehungsweise *ExportBed* dem Import-Abschnitt hinzugefügt werden.

Wenn bestimmte Objekte in mehreren Aktionen oder Bedingungen benutzt werden sollen, so müssen entsprechende Vorkehrungen in denselben Klassen getroffen werden. Solche Objekte müssen sie als öffentliche Klassenvariablen (`public static`) deklariert werden.

Es sei hier nochmals darauf hingewiesen, dass Java-Aktionen und Bedingungen in der Simulation nicht ausgeführt beziehungsweise getestet werden. Aus diesem Grund, und um nicht der Versuchung zu erliegen, das Statechart zu umgehen, sollten Zugriffe auf das Statechart nach Möglichkeit vermieden und durch interne Aktionen und Bedingungen ersetzt werden.

Sowohl in Bedingungen als auch Aktionen wird durch die Methode **variable(*String*)** der Zugriff auf Variablen des Statechart ermöglicht. Sie gibt eine Variable (*score.Simulator.Variable*) zurück; wie mit einer solchen gearbeitet werden kann, ist in der folgenden Tabelle aufgeführt.

set(Wert) Setzt die Variable auf den angegebenen Wert. Dieser kann vom Typ *int*, *double* oder *String* sein; die Variable wird auf diesen Typ konvertiert. Es ist auch möglich, eine weitere Variable zu übergeben; in diesem Fall wird deren Wert übernommen.

inc(Wert) Erhöht die Variable um den angegebenen Wert. Dieser kann vom Typ *int*, *double* oder *String* sein; wird eine Zahl übergeben, wird die Variable auf den neuen Typ konviertiert, im letzten Fall wird der übergebene String angehängt. Es ist auch möglich, eine weitere Variable zu übergeben; hierbei wird entsprechend dem Typ der übergebenen Variable verfahren.

getString() Gibt eine String-Repräsentation dieser Variablen zurück.

getInt() Gibt den Wert der Variablen als *int* zurück, falls dies ein gültigen Wert ergibt.

getDouble() Gibt den Wert der Variablen als *double* zurück, falls dies ein gültigen Wert ergibt.

compare(Operator, Wert) Als Operator sind die öffentlichen Konstanten (`public static final int`) der Klasse *Variable* möglich: `EQUAL`, `UNEQUAL`, `LESS`, `GREATER`, `LESS_OR_EQUAL` und `GREATER_OR_EQUAL`. Für den Wert sind *int*, *double*, *String* oder andere Variablen erlaubt; der Vergleich erfolgt dem übergebenen Typ entsprechend.

In Aktionen sind darüber hinaus die folgenden Methoden vorhanden:

throwEvent(*String*) Lediglich in Aktionen kann mit dieser Methode ein neues Ereignis erzeugt werden. Der übergebene String, der den Namen des Ereignisses angibt, dient hierbei als Kennzeichner.

visible(*int*) invisible(*int*) Diese Methoden machen Dialogkomponenten sichbar beziehungsweise unsichtbar. Sie erwarten den Identifier des Dialogelements als Parameter. Welche Wirkung ein solcher Aufruf auf die unterschiedlichen Typen von Dialogelementen hat, wird in Abschnitt 5.8 beschrieben.

on(*int*) off(*int*) Diese Methoden schalten Dialogkomponenten ein beziehungsweise aus. Sie erwarten den Identifier des Dialogelements als Parameter. Welche Wirkung ein solcher Aufruf auf die unterschiedlichen Typen von Dialogelementen hat, wird in Abschnitt 5.8 beschrieben.

In Bedingungen sind die folgenden Abfrage möglich:

in(*int*) Diese Methode gibt `true` zurück, wenn der Zustand mit dem übergebenen Identifier momentan aktiv ist; diese Abfrage entspricht der atomaren internen Bedingung `in?(S)`.

hasHist(*int*) Diese Methode gibt `true` zurück, wenn der Zustand mit dem übergebenen Identifier momentan eine History besitzt; diese Abfrage entspricht der atomaren internen Bedingung `hasHist?(S)`.

5.12 Erweitern des Programms um neue Dialogkomponenten

Um das Programm um weitere Dialogkomponenten zu erweitern, erstellt man eine neue Klasse, die von DialogKomponente erbt. Theoretisch kann diese sofort benutzt werden, jedoch fehlen noch die Einträge in der Benutzeroberfläche und der Sprache, um die neue Komponente auszuwählen bzw. zu laden und zu speichern. Danach ist die Komponente voll einsatzbereit.

5.13 Der Verifikator

5.13.1 Überblick

Mit dem Verifikator lässt sich das konstruierte Statechart auf Korrektheit in verschiedenen Punkten überprüfen. Einige Probleme werden als Fehler gemeldet; andere anscheinende Unstimmigkeiten können vom Benutzer durchaus gewollt sein und führen

nicht notwendigerweise zu Fehlern. In diesen Fällen gibt der Verifikator Warnmeldungen aus.

Vor dem Start dieses Tests kann durch die Auswahl der Optionen angegeben werden, welche Eigenschaften des Statecharts untersucht werden sollen. Es ist empfehlenswert, zunächst alle Punkte zu überprüfen, und die Auswahl lediglich im weiteren Verlauf der Testphase einzuschränken, etwa um die Anzahl der Warnungen in einem überschaubaren Rahmen zu halten.

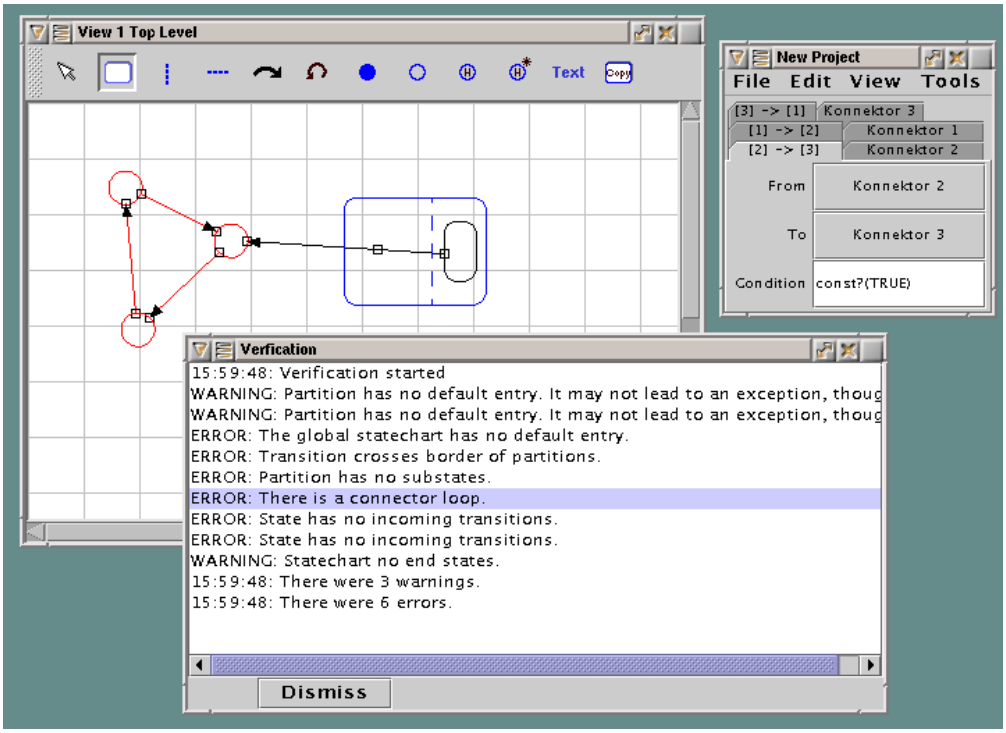


Abbildung 5.20: Der Verifikator nach einem Durchlauf

Der Test wird gestartet, wenn der Verifikator im Menü aufgerufen wird. In einem neuen Fenster gibt *SCed* dann Fehlermeldungen und Warnungen aus, die in diesem Abschnitt genauer erläutert werden. Siehe auch Abbildung 5.20.

Wenn sich die Meldungen auf Zustände, Pfeile oder Dialogelemente beziehen, so können sie in diesem Fenster angeklickt werden, woraufhin das oder die zugehörigen Elemente des Statecharts markiert, das heißt im Inspektor und im Editor angezeigt werden. Gibt es keine solchen Elemente, wird der Inspektor nach Anklicken der Meldung `Nothing` anzeigen.

5.13.2 Die einzelnen Tests

Über die Test-Optionen lassen sich verschiedene Tests auswählen oder überspringen. Siehe auch Abbildung 5.21. In diesem Abschnitt sollen die überprüfbaren Eigenschaften und die dabei auftretenden Fehlermeldungen und Warnungen kurz beschrieben werden.

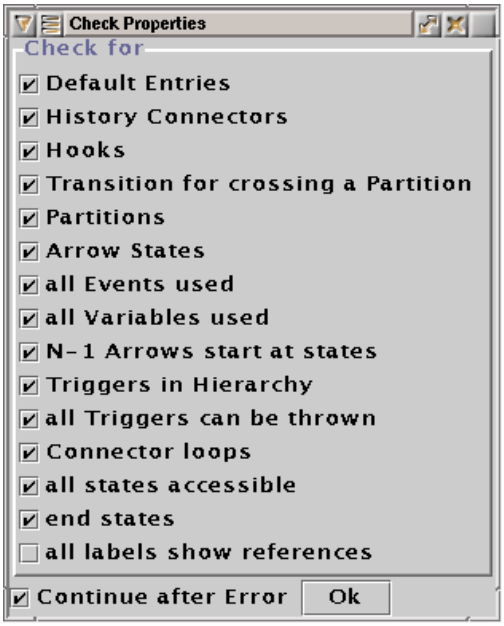


Abbildung 5.21: Das Auswahlfenster für den Verifikator

Continue after Error

Diese Einstellung beeinflusst das Verhalten des Verifikators, wenn Fehler gefunden wurden. Wird sie abgewählt, so bricht der Testlauf nach der ersten Eigenschaft ab, bei deren Überprüfung Fehler auftraten. Warnungen führen hierbei nicht zu einem Abbruch.

Default Entries

Wird dieser Test angewählt, so werden die Default-Einstiege des gesamten Statecharts getestet. Insbesondere wird jeder OR-Zustand und das gesamte Statechart daraufhin untersucht, ob sie genau einen Default-Einstieg besitzen. Außerdem muss jeder Default-Einstieg genau einen ausgehenden 1-1 Pfeil besitzen, der weder Ereignisse, Bedingungen noch Aktionen besitzen darf und auf einen Zustand enden muss, der im selben OR-Zustand wie der Default-Einstieg liegt.

Die folgenden Meldungen können ausgegeben werden:

[...] has no default entry. Diese Meldung kann je nach Bezug mit „State“, „Partition“ oder „The global Statechart“ beginnen. Sie kann entweder einen Fehler anzeigen oder nur eine Warnung sein, je nach dem, ob der Default-Einstieg wirklich gebraucht wird. Ein OR-Zustand etwa, der nur dadurch betreten wird, dass einer seiner Unterzustände aktiviert wird, braucht zunächst keinen Default-Einstieg. Die Warnung wird dennoch ausgegeben, da sich eventuell durch Weiterentwicklung des Statecharts diese Notwendigkeit ergeben könnte.

[...] has more than one default entry. Diese Fehlermeldung wird dann angezeigt, wenn in einem OR-Zustand, einer Partition oder der obersten Ebene des globalen Statecharts mehr als ein Default-Einstieg angegeben ist. Entsprechend beginnt diese Meldung mit „State“, „Partition“ oder „The global Statechart“.

Default entry does not lie within an OR-state. Wenn ein Default-Einstiegs-Konnektor in einen atomaren Zustand gelegt wird, wird diese Fehlermeldung ausgegeben, da der Bezug dieses Konnektors dann nicht klar ist.

Default entry has no/more than one outgoing arrow. Ein Default-Konnektor darf nur genau einen ausgehenden Pfeil besitzen. Da diese Pfeile weder Ereignisse noch Bedingungen haben dürfen, wäre sonst nicht eindeutig, welcher Zustand gemeint ist.

Default entry connector needs outgoing arrow of type 1-1. Wenn der Pfeil am Default-Einstieg vom Typ 1-N oder N-1 ist, wird diese Fehlermeldung ausgegeben. Beide Typen sind nicht erlaubt. Sollen durch den ausgehenden 1-N Pfeil mehrere Partitionen betreten werden, so ist es besser, jeder Partition einen eigenen Default-Einstieg zuzuweisen.

Default entry connector needs outgoing arrow without event, condition or action. Ein Pfeil, der von einem Default-Einstieg ausgeht, darf weder durch ein Ereignis noch eine Bedingung eingeschränkt sein. Außerdem darf sich an ihm keine Aktion befinden; eine solche Aktion lässt sich stattdessen dem Zielzustand als Aktion beim Aktivieren zuordnen.

Default entry must mark a substate of its own containing state. Der von einem Default-Konnektor ausgehende Pfeil muss auf einem Zustand enden, der direkt in dem Zustand liegt, in dem auch der Default-Konnektor liegt. Es ist nicht möglich, auf diese Weise mehrere Hierarchieebenen des Statecharts zu überqueren; stattdessen sollte in jeder Ebene ein eigener Default-Einstieg gesetzt werden.

History connectors

History connector’s outgoing arrows must not lead to this end state. Pfeile, die von einem History-Konnektor ausgehen, dürfen nur auf Zuständen enden, die im selben OR-Zustand (oder in derselben Partition) liegen, wie der History-Konnektor selbst.

History connector may only have outgoing arrows of type 1-1. Die Pfeile, die von einem History-Konnektor ausgehen, dürfen nur vom Typ 1-1 sein. Sollen mehrere Partitionen in den Übergang miteinbezogen werden, so sollte für jede ein eigener History-Konnektor verwendet werden.

Hooks

Some hooks lie within states that are unrelated to their transition. Die Zwischenpunkte eines Pfeils (Hooks) dürfen nur in Zuständen liegen, die von diesem Übergang

entweder deaktiviert oder aktiviert werden oder den Pfeil komplett umfassen. Ist dies nicht der Fall, wird diese Fehlermeldung ausgegeben. Abbildung 5.22 zeigt eine solche Situation: Der Hook in Zustand C ist ungültig, da dieser Zustand von einem Übergang von Zustand A auf Zustand B unberührt bleibt.

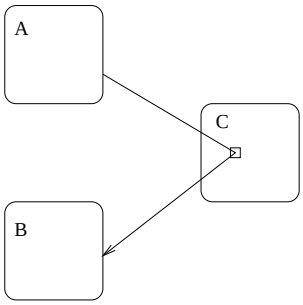


Abbildung 5.22: Ein ungültig platzierter Hook

Transition for crossing a Partition

Transition crosses border of partitions. Diese Fehlermeldung wird ausgegeben, wenn ein Pfeil die Trennlinie von Partitionen eines AND-Zustands schneidet. Dies ist für *SCed* der Fall, wenn benachbarte Hooks in unterschiedlichen Partitionen liegen.[11]

Partitions

Partition has no substates. In Statecharts sind keine leeren Partitionen erlaubt. Werden solche gefunden, wird diese Fehlermeldung ausgegeben.

Arrow States

Dieser Test überprüft die Lage von Start- und Endzuständen aller Verbindungen auf ihre Lage zueinander. Fehler können dabei nur an Pfeilen vom Typ 1-N oder N-1 auftreten.

States must not appear as start point more that once per transition. Ein Zustand wurde mehrfach als Startzustand eines Übergangs angegeben. Es reicht in diesem Fall aus, ihn nur einmal zu benutzen.

Some of the transition's start states lie above an other. Der Übergang beginnt bei mehreren Zuständen, die in der Hierarchie übereinander liegen. In diesem Fall reicht es, den untersten anzugeben, da sämtliche Oberzustände notwendigerweise ebenfalls aktiv sein müssen, wenn dieser aktiv ist.

[11] vgl. hierzu Abbildung 3.5 in Abschnitt 3.2.1

States must not appear as end points more than once per transition. Ein Zustand wurde mehrfach als Endzustand eines Übergangs angegeben. Es reicht in diesem Fall aus, ihn nur einmal zu benutzen.

Some of the transition's end states lie above each other. Der Übergang endet bei mehreren Zuständen, die in der Hierarchie übereinander liegen. In diesem Fall reicht es, den untersten anzugeben, da sämtliche Oberzustände notwendigerweise ebenfalls aktiviert werden, wenn dieser aktiviert wird.

All Events used

The following events aren't checked in any transition. Dieser Test kann höchstens zu Warnungen führen. Er überprüft, ob alle Ereignisse, die in der Ereignisliste eingetragen sind, irgendwo als Auslöser eines Übergangs benutzt werden. Ausgeschlossen sind hiervon die automatisch erzeugten Ereignisse `enter.()` und `exit.()`. Diese Meldung wird auch für Ereignisse ausgegeben, die nie ausgelöst werden. In diesem Fall beginnt die Zeile mit dem Zusatz `(unthrown)`.

Beim Anklicken werden alle Elemente des Statecharts markiert, die das betreffende Ereignis auslösen.

All Variables used

The following variables are set but never checked. Mit diesem Test wird überprüft, ob alle Variablen, denen irgendwo ein Wert zugewiesen wird, auch in einer Bedingung abgefragt werden. Ausgeschlossen sind hierbei die Variablen, die Dialogelementen zugeordnet sind (und mit `DLG` beginnen). Diese Meldung wird auch für Variablen ausgegeben, die in Aktionen gesetzt werden, die keinem Element des Statecharts zugeordnet sind. In diesem Fall beginnt die Zeile mit dem Zusatz `(in uncalled action)`.

Beim Anklicken werden alle Elemente des Statecharts markiert, die den Wert der Variablen setzen.

The following variables are checked but never set. Mit diesem Test wird die Umkehrung des letztes Tests durchgeführt, indem überprüft wird, ob allen Variablen, die in einer Bedingung benutzt werden, auch irgendwo ein Wert zugewiesen wird. Ausgeschlossen sind hierbei wieder die Variablen, die Dialogelementen zugeordnet sind (und mit `DLG` beginnen). Diese Meldung wird auch für Variablen ausgegeben, die in Aktionen getestet werden, die keinem Element des Statecharts zugeordnet sind. In diesem Fall beginnt die Zeile mit dem Zusatz `(in uncalled action)`.

Beim Anklicken werden alle Elemente des Statecharts markiert, die den Wert der Variablen nutzen.

N-1 Arrows start at states

N-1 arrow starts at non-state. Pfeile vom Typ N-1 dürfen nur an Zuständen beginnen. Diese Fehlermeldung wird ausgegeben, wenn ein Pfeil an einem Konnektor beginnt.

Triggers in Hierarchy

The following events trigger transitions on stacked levels. Diese Warnung wird ausgegeben, wenn von einem Zustand und seinen Oberzuständen mehrere Pfeile ausgehen, die vom selben Ereignis ausgelöst werden. Hierbei wird nicht überprüft, ob diese Übergänge sich möglicherweise durch ihre Bedingungen gegenseitig ausschließen.

Connector loops

There is a connector loop. Wenn ein Konnektor durch einen oder mehrere Übergänge wieder auf sich selbst führen kann, so wird diese Fehlermeldung ausgegeben. Hierbei wird nicht überprüft, ob ein solcher Rundlauf durch angegebene Ereignisse oder Bedingungen ausgeschlossen ist. Es ist jedoch in so einem Fall besser, diesen Kreis zu einem einzigen Konnektor zusammenzufassen.

All states accessible

In diesem Test wird überprüft, ob alle Zustände und Konnektoren eingehende Verbindungen besitzen. Dabei wird bei Zuständen auch darauf geachtet, ob möglicherweise Verbindungen auf Unterzustände führen.

Connector has no incoming transitions. Ein Konnektor, der kein Default-Entry-Konnektor ist, hat im Statechart nur mit eingehenden Verbindungen eine sinnvolle Aufgabe. Für alle in diesem Sinne überflüssigen Konnektoren wird diese Fehlermeldung ausgegeben.

State has no incoming transitions. Wenn ein Zustand weder direkt noch über Verbindungen, die auf Unterzustände führen, betreten werden kann, wird diese Fehlermeldung ausgegeben. Möglicherweise wurde lediglich vergessen, den Default-Einstieg des darüberliegenden OR-Zustands entsprechend zu setzen.

End states

End state has outgoing transitions. Wird ein Endzustand betreten, wird dadurch das System angehalten; es ist also nicht sinnvoll, von einem solchen Zustand ausgehende Verbindungen einzugeben. Geschieht dies trotzdem, wird diese Fehlermeldung ausgegeben.

End state has actions on deactivation. Da ein Endzustand nicht verlassen werden kann, macht es auch keinen Sinn, Aktionen für diesen Fall anzugeben. Geschieht dies trotzdem, wird diese Fehlermeldung ausgegeben.

Zu beachten ist hierbei das Folgende: ist als einzige Aktion eine Throw-Aktion mit einem Ereignis angegeben, dessen Text mit „exit.(“ beginnt, so geht der Verifikator davon aus, dass es sich um die automatisch generierte Aktion handelt. Dieser Fall ruft keine Fehlermeldung hervor.

State cannot be left and is not marked as end state. Wenn ein Zustand, der nicht als Endzustand markiert ist, nicht verlassen werden kann, wird diese Warnung ausgegeben. Hierbei werden auch von Unter- und Oberzuständen ausgehende Verbindungen beachtet. Es wird jedoch nicht überprüft, ob ein solcher Zustand eventuell Endzustände enthält und die Konstruktion daher konsistent ist.

Statechart has no end states. Diese Warnung zeigt an, dass im gesamten Statechart kein einziger Zustand als Endzustand markiert ist. Ein solches System kann nur von außen beendet werden. Da dies aber gewünscht sein kann, ist diese Meldung nur eine Warnung.

All labels show references

Arrow label differs from actual values of that arrow. An Pfeilen können Texte angebracht werden, an denen die Ereignisse, Bedingungen und Aktionen angezeigt werden, die zu diesem Pfeil gehören. Es wird in vielen Fällen sinnvoll sein, den Text dieses Bezeichners nachträglich von Hand zu modifizieren, beispielsweise zu kürzen. Diese Warnung wird ausgegeben, wenn ein derart geänderter Text gefunden wird. Dieser Test ist jedoch in der Grundeinstellung nicht angewählt.

It's a miracle we need.
(Queen)

Kapitel 6

Funktionsweise der Module

6.1 Überblick

In diesem Kapitel sollen die einzelnen Module von *SCed* kurz in ihrer Funktionsweise vorgestellt werden. Es soll dadurch helfen, das Programm zu verstehen, ohne die gesamte Programmdokumentation durcharbeiten zu müssen. Einen Überblick über die Zusammenarbeit der Pakete gibt Abbildung 6.1. Aufgabe dieses Kapitels ist jedoch nicht, das Arbeiten mit dem Programm zu erläutern. Für eine derartige Einführung sei auf das Benutzerhandbuch in Kapitel 5 verwiesen.

Zunächst wird beschrieben, auf welche Weise das Statechart modelliert wurde, und wie die Dialogkomponenten im Speicher gehalten werden. Die darauf folgenden Abschnitte beschäftigen sich mit einzelnen Modulen, insbesondere den beiden Editoren, dem Simulator und dem Quelltextexport.

Das Kapitel schließt mit einem Ausblick, wie das vorliegende Programm erweitert werden könnte, und erwähnt kurz, mit welchem Aufwand diese Erweiterungen verbunden wären.

6.2 Die Datenstruktur des Statecharts

6.2.1 Überblick

Die Spezifikationen des Statecharts werden in zwei parallelen Modellen gehalten, die unterschiedliche Daten speichern. Diese beiden Modelle sind das Logische Statechart-Modell (LogSCM) und das Grafische Statechart-Modell (GrafSCM). Der Schwerpunkt des logischen Modells liegt auf der Struktur des Statecharts: welcher Zustand atomar oder komplex ist und welche Unterzustände enthält, oder bei welchen Zuständen Übergänge beginnen und enden und welche Ereignisse, Bedingungen und Aktionen ihnen zugeordnet sind. In diesem Modell werden Partitionen durch OR-Zustände modelliert, wie es auf Seite 58 vorgeschlagen wurde. Das grafische Modell hält die Daten, die für die Darstellung des Statecharts relevant sind: die Größe der Zustände und ihre Position, aus welchen Abschnitten die Pfeile bestehen, und wo die Zwischenpunkte (*Hooks*[1]) liegen. Die Konstruktion dieser beiden Modelle ist in den Abbildungen 6.2 und 6.3 dargestellt.

[1] vgl. Seite 61

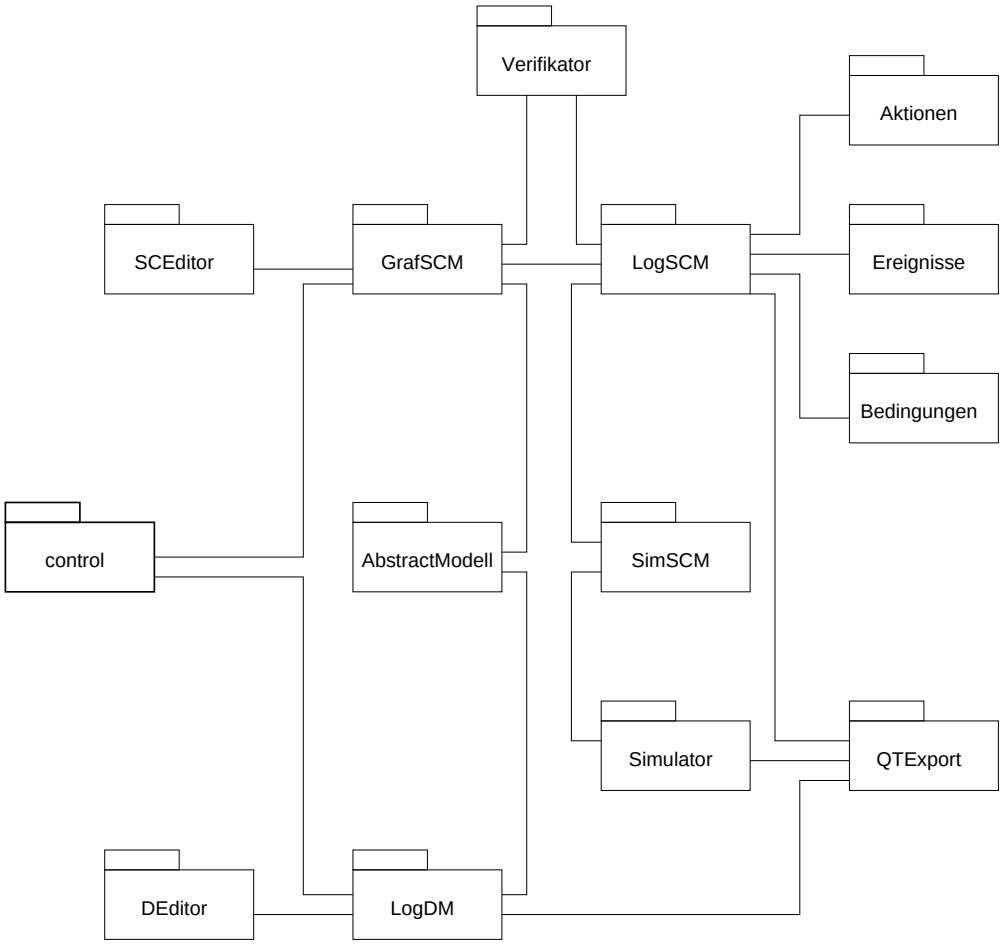


Abbildung 6.1: Überblick über die Pakete

Während die Simulation läuft, existiert noch ein weiteres Modell, das Simulationsmodell (SimSCM). Dieses Modell verwaltet die Information, welche Zustände momentan aktiv sind, und aktiviert beziehungsweise deaktiviert die Ziel- und Startzustände von Übergängen sowie die dafür notwendigen Ober- und Unterzustände.[2] Auch im exportierten Quelltext wird prinzipiell das Simulationsmodell benutzt; allerdings werden die Übergänge aus dem logischen Modell übernommen. Dieses Modell ist in Abbildung 6.4 dargestellt; Abschnitt 6.5 beschäftigt sich mit einer detaillierten Beschreibung des Simulationsablaufes und der Zusammenarbeit von Simulator und Simulationsmodell.

Die folgenden Abschnitte beschäftigen sich mit einer etwas detaillierteren Vorstellung dieser drei Modelle.

6.2.2 Das Logische Statechart-Modell

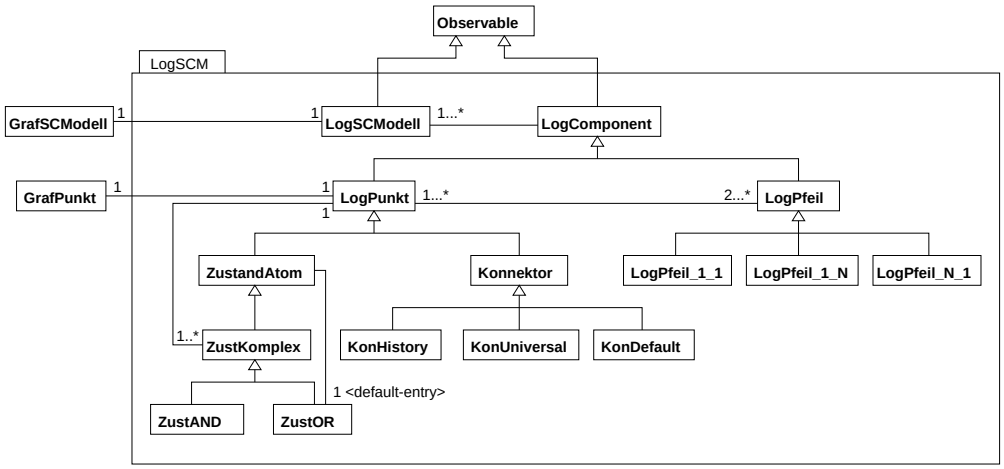


Abbildung 6.2: Klassenüberblick des Pakets *LogSCM*

Das Logische Statechart-Modell ist das grundlegendste der drei Modelle. Es enthält die wichtigsten Informationen des Statecharts und ist prinzipiell ohne die beiden anderen Modelle denkbar.

Zu den einzelnen Klassen sind die folgenden Anmerkungen zu machen:

LogSCModell Diese Klasse ist die Oberklasse des Logischen Statechart-Modells. Sie implementiert das Interface Observer und beobachtet dadurch alle ihre Komponenten. Sie enthält einen virtuellen (also in der Sicht auf das Statechart nicht dargestellten) Oberzustand, unter den sämtliche Elemente des Statecharts untergeordnet werden.

LogComponent Diese Klasse ist die Grundklasse für alle logischen Punkte und Pfeile; sie wird von der Oberklasse des logischen Modells beobachtet, so dass Änderungen an diese weitergegeben werden können.

[2] Eine genaue Beschreibung dieses Verfahrens findet sich in Abschnitt 6.5.3.

LogPunkt Diese Klasse ist die abstrakte Oberklasse aller Punkte im Statechart, das heißt aller Stellen, an denen logische Pfeile beginnen oder enden können. Diese Übergänge werden im LogPunkt gehalten. Außerdem besitzt jeder LogPunkt einen Oberzustand.

ZustandAtom Der atomare Zustand enthält die meisten Informationen eines Zustands. Hier werden die Aktionen gespeichert, die beim Aktivieren und Deaktivieren dieses Zustands ausgelöst werden sollen. Wird ein solcher Zustand erschaffen, werden die Aktionen automatisch so initialisiert, dass beim Aktivieren das Ereignis `enter.(S)` und beim Deaktivieren das Ereignis `exit.(S)` ausgelöst werden.

ZustKomplex Der komplexe Zustand erweitert den atomaren Zustand in dem Sinne, dass er Unterzustände enthalten kann. Er ist jedoch noch abstrakt und muss entweder als OR- oder AND-Zustand erweitert werden.

ZustAND Diese Klasse erweitert den komplexen Zustand nur um spezielle Methoden, aber nicht um weitere Daten.

ZustOR Ein OR-Zustand kann die Information enthalten, welcher seiner Unterzustände als Defaulteinstieg markiert ist.

Konnektor, KonUniversal, KonHistory Diese Klassen erweitern den logischen Punkt entsprechend. Sie unterscheiden sich von den Zuständen darin, dass sie nicht aktiviert werden können, und deshalb auch keine Aktionen enthalten.

KonDefault Diese Klasse ist lediglich während der Konstruktion des Statecharts als logisches Gegenstück zum grafischen Defaulteinstiegs-Konnektor notwendig. Für die Simulation wird stattdessen der entsprechende Unterzustand eines jeden OR-Zustands als Defaulteinstieg markiert.

LogPfeil Die abstrakte Klasse LogPfeil wird von den drei spezialisierten Pfeiltypen erweitert. Dementsprechend sind Anfangs- und Endpunkte des Übergangs auch erst in den Unterklassen enthalten. Allen gemeinsam sind jedoch ein auslösendes Ereignis, eine Bedingung und eine Aktion; diese sind daher bereits in LogPfeil enthalten.

LogPfeil_1_1, LogPfeil_1_N, LogPfeil_N_1 Je nach Typ des Übergangs enthalten diese Klassen einen einzelnen LogPunkt oder einen Vektor von LogPunkten als Anfangs- und Endpunkte.

6.2.3 Das Grafische Statechart-Modell

Die Klassen des Logischen Statechart-Modells haben alle eine Entsprechung im grafischen Modell, allerdings werden dabei mehrere logische Klassen in einer grafischen Klasse zusammengefasst.

So gibt es im GrafSCM keine eigene Klasse für OR-Zustände; diese finden sich zusammen mit den atomaren Zuständen in der Klasse GrafZustand. Die Klasse für AND-Zustände wird im GrafSCM von diesem gemeinsamen Zustand abgeleitet und lediglich um die Information über die Partitionsgrenzen (*Partitioners*) erweitert.

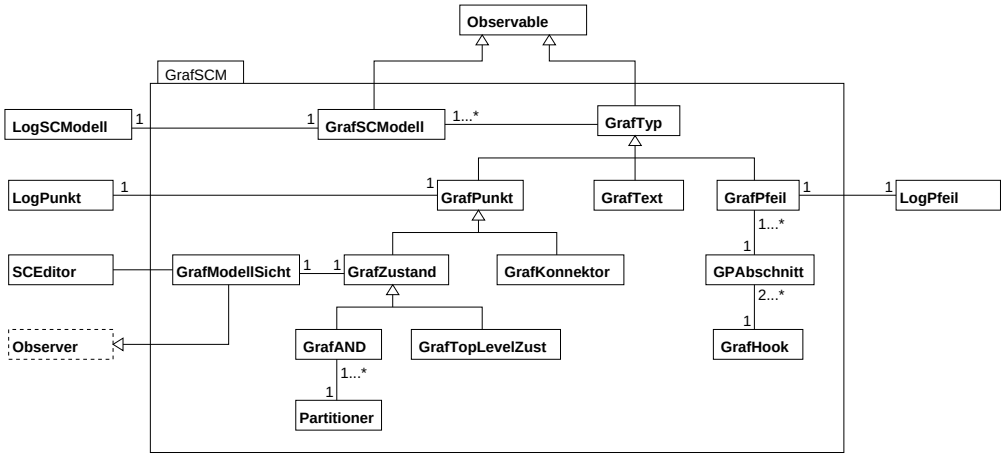


Abbildung 6.3: Klassenüberblick des Pakets *GrafSCM*

Andererseits ist die Darstellung der Übergänge im grafischen Modell aufwendiger als im logischen, so dass hier neue Klassen hinzukommen: die Abschnitte eines Pfeils und die sogenannten Hooks, das sind die Eckpunkte eines Pfeils und gleichzeitig die Stellen, an denen neue Abschnitte angekoppelt werden können.

Besonders erwähnenswert ist die Verwaltung der Koordinaten als lokale Koordinaten, das heißt die Offsets der Zustände sind bezüglich ihres direkten Oberzustands angegeben. Das ermöglicht das einfache Verschieben eines Zustands mitsamt seiner Unterzustände, oder auch das einfache Beschränken der Ansicht auf einen einzigen Zustand.

Zu den einzelnen Klassen sollten die folgenden Dinge gesagt werden:

GrafSCModell Analog zum LogSCModell ist dies die Klasse, die den grafischen virtuellen Oberzustand enthält. Sie beobachtet ihre Komponenten, um Änderungen an den Benutzer weitergeben zu können.

GrafModellSicht Eine Sicht auf das grafische Modell bezieht sich immer auf einen Zustand. Für jeden Zustand sind Vergrößerung und Darstellungstiefe unabhängig wählbar, und diese Angaben werden in der zugehörigen Sicht gespeichert, so dass sie beim nächsten Aufruf auf die vorherigen Werte gesetzt werden können.[3]

GrafTyp Diese Klasse erweitert die abstrakte Klasse MarkableTyp aus dem abstrakten Modell zur Oberklasse für sämtliche grafischen Komponenten. Alle Unterklassen von GrafTyp können vom Benutzer angeklickt und im Inspektor dargestellt werden.

GrafPunkt Der GrafPunkt erweitert den GrafTyp analog zum LogPunkt im logischen Modell: er ist die abstrakte Oberklasse für alle Punkte des Statecharts, an denen Übergänge beginnen oder enden können. Die entsprechenden grafischen Pfeile

[3] vgl. die Beschreibung der Ansichten auf Seite 60.

werden in dieser Klasse gespeichert. Außerdem besitzt ein grafischer Punkt eine Größe und einen Offset bezüglich der linken oberen Ecke seines Oberzustands.

GrafZustand Der grafische Zustand entspricht den atomaren Zuständen und den OR-Zuständen des logischen Modells. Er kann Unterzustände und eine eigene Sicht enthalten. Wird ein neuer Unterzustand eingefügt, so wird das logische Gegenstück gegebenenfalls von einem atomaren auf einen OR-Zustand umgestellt. Unterzustände erhalten beim Einfügen jeweils einen Offset, der von der linken oberen Ecke ihres direkten Oberzustands gerechnet ist.

GrafAND Dies ist die beschriebene Erweiterung des grafischen Zustands um Partitioners. Außerdem sind für AND-Zustände andere Methoden zur Berechnung der Tiefe und Sichtbarkeit notwendig.

Partitioner Auch diese Klasse erbt von GrafTyp, da sie vom Benutzer markiert werden kann. Ein Partitioner ist eine Trennlinie zwischen zwei Partitionen eines AND-Zustands. Ihre beiden Endpunkte sind dabei bezüglich der linken oberen Ecke des AND-Zustands angegeben.

GrafTopLevelZust Obwohl der virtuelle Oberzustand in den Sichten nicht auftaucht, ist er im Prinzip ein OR-Zustand. Dennoch müssen einige Funktionen überschrieben werden, um dieser Sonderrolle gerecht zu werden.

GrafKonnektor Diese Klasse fasst alle logischen Konnektoren zusammen. Sie erweitert den GrafPunkt ähnlich wie GrafZustand, nur dass keine Unterzustände erlaubt sind und der Konnektor nicht durch ein abgerundetes Rechteck sondern einen Kreis dargestellt wird. Dementsprechend sind andere Methoden nötig, um die Positionen der Start- und Endpunkte von Übergängen zu berechnen.

GrafPfeil Der grafische Pfeil entspricht allen drei Typen von logischen Pfeilen und erweitert die Klasse GrafTyp. Er besitzt einen Vektor von Startabschnitten und einen von Endabschnitten sowie ein logisches Äquivalent. Bei einem 1-1 Übergang sind keine Endabschnitte vorhanden.

GPAbschnitt Diese Klasse stellt den oben erwähnten Abschnitt eines grafischen Pfeils dar. Sie erbt von der Java-Klasse Vector und ist damit als Vektor von GrafHooks implementiert.

GrafHook Die Hooks sind die Ecken eines grafischen Pfeils; sie besitzen keine Entsprechung im logischen Modell. Hooks gibt es in fünf verschiedenen Formen: als Start- beziehungsweise Endhook, als normaler mittlerer Hook und als Hook, an dem mehrere Abschnitte zusammenkommen oder von dort ausgehen.

GrafText Mit dieser Klasse können Texte zum Statechart hinzugefügt werden. Wenn sich der Text auf einen GrafPfeil bezieht, wird hierin auch verwaltet, ob der angezeigte Text mit den Eigenschaften des Pfeil übereinstimmt.

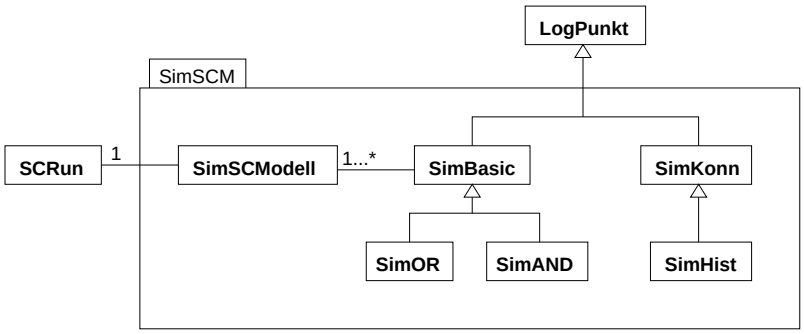


Abbildung 6.4: Klassenüberblick des Pakets *SimSCM*

6.2.4 Das Simulations- und Quelltextmodell

Das Simulationsmodell wird erst beim Start einer Simulation aus dem logischen Modell erschaffen. Anders als in den beiden anderen Modellen gibt es hier nur Zustände, keine Übergänge. Diese werden während der Simulation von der Steuerklasse der Simulation aus dem logischen Modell gelesen und ausgeführt.

Auch beim Quelltextexport wird dieses Modell benutzt. Die in diesem Modell vorhandene Klasse der Konnektoren wird nur für diesen Fall benutzt. Die Verbindungen werden aus dem logischen Modell übernommen.

Das hier beschriebene Modell ist jedoch kein aktives Modul; es dient wie die anderen beiden Modelle primär der Datenspeicherung. Es sorgt allerdings dafür, dass Ober- und Unterzustände so aktiviert beziehungsweise deaktiviert werden, dass der Zustand des Statecharts konsistent ist. Die eigentliche Durchführung der Simulation beziehungsweise des ausführbaren Programms obliegt der Klasse *SCRun*, die in Abschnitt 6.5.2 genauer beschrieben wird.

Bei den Klassen des Modells sei auf folgende Dinge hingewiesen:

SimSCModell Wie bei den anderen Modellen auch enthält diese Klasse den Oberzustand. Außerdem ist über diese Klasse der Zugriff auf die Simulationszustände möglich. Insbesondere können so die Bedingungen testen, ob bestimmte Zustände aktiv sind.

SimBasic Ein SimBasic ist ein aktivierbarer Zustand. Er hat eine logische Entsprechung, die mindestens ein atomarer Zustand ist. Während die Übergänge berechnet werden, gibt es zwei Vektoren von besonders markierten Zuständen: diejenigen, die gerade noch aktiv sind, und diejenigen, die gerade aktiv werden.

SimOR Mit dieser Klasse wird die Klasse SimBasic zu einem OR-Zustand erweitert. Sie erhält Unterzustände, von denen einer als Defaulteinstieg markiert ist, sowie einen aktiven Unterzustand und einen History-Zustand, das heißt den Zustand, der als letztes aktiv war.

SimAND Diese Klasse erweitert den atomaren Zustand im Sinne eines AND-Zustands. Die Methoden zur Aktivierung und Deaktivierung sind entsprechend überschrieben. Der SimAND enthält auch einen Vektor aller AND-Zustände, die

gerade aktiviert wurden, um diese nach einem Zeitschritt nachführen zu können. Für eine genauere Beschreibung dieses Verfahrens sei auf Abschnitt 6.5.3 hingewiesen.

SimConn Diese Klasse wird nur im exportierten Quelltext benötigt und stellt dabei einen Konnektor dar. Er unterscheidet sich von der Klasse SimBasic darin, dass er nicht aktiviert werden kann.

SimHist Auch diese Klasse wird in der Simulation selbst nicht benötigt; für den exportierten Quelltext modelliert sie einen (entwder einfachen oder Deep-) Historykonnektor.

6.3 Das logische Dialogmodell

Die Daten über die mit dem Statechart interagierenden Dialoge werden im logischen Dialogmodell gehalten. Um diese Daten zu bearbeiten, steht der Dialog-Editor zur Verfügung.

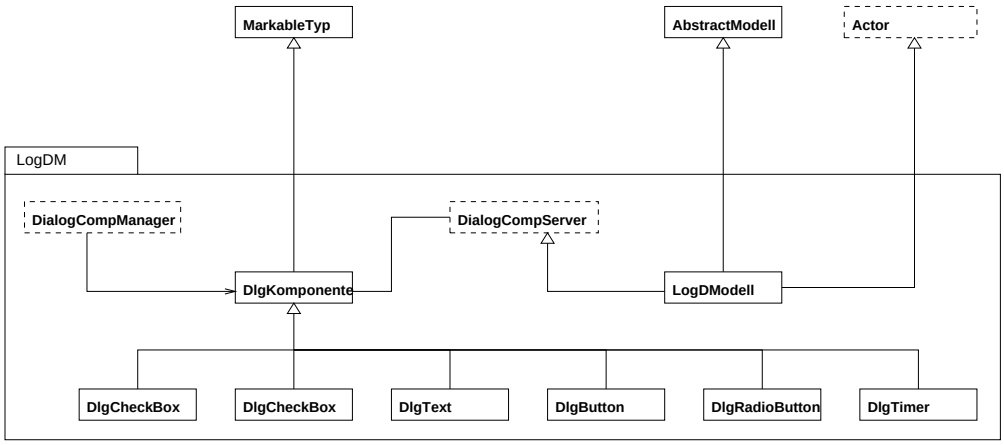


Abbildung 6.5: Aufbau des logischen Dialogmodells

Das logische Dialogmodell (LogDM) besteht aus den Klassen für die logischen Dialogkomponenten (DlgCheckbox, DlgPopup, DlgRadiobutton, DlgButton, DlgText und DlgTimer), einer Oberklasse für diese Komponenten (DlgKomponente), einer Klasse, die diese Komponenten hält (LogDModell) und zwei Interfaces, die den Zugriff auf die Komponenten regeln (DialogCompManager und DialogCompServer). Den hierarchischen Überblick gibt Abbildung 6.5.

Das LogDModell hält die logischen Dialogkomponenten. Ein Zugriff ist von außen über das DialogCompServer-Interface möglich. Dieses liefert zu einer ID die dazugehörige DlgKomponente.

Über den DialogCompManager ist der Zugriff auf DlgKomponenten möglich. Mit diesem Interface können die Komponenten sichtbar/unsichtbar oder ein/aus geschaltet werden. Dieses Interface wird von der Statechart-Klasse implementiert.

6.4 Der Editor

Der Statechart-Editor

Der Statechart-Editor (SCEditor) liefert das grafische Benutzerinterface um das grafische Statechart-Modell (GrafSCM) zu bearbeiten. Die Hauptklasse ist hier SCEditor. Diese erstellt das Fenster mit den Werkzeugleisten und enthält die Zeichenfläche vom Typ SCCanvas. Es werden diverse Klassen zur Darstellung von Editiervorgängen benötigt, wie alle Rubber-Klassen, sowie das TranslationWindow und der Painter, über den die Sicht ihre Daten zeichnet. Einen Überblick gibt Abbildung 6.6.

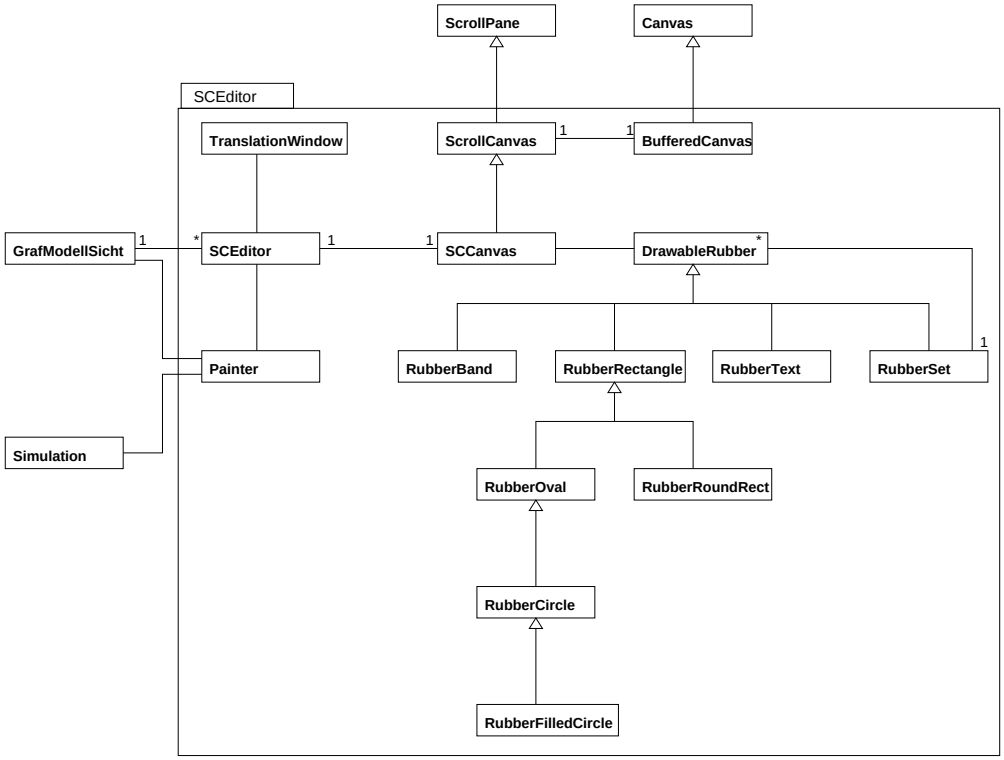


Abbildung 6.6: Überblick über die Klassen des Statechart-Editors

Der Editor ist eines der Kernstücke unserer Implementation. Mit ihm wird das Statechart grafisch erstellt beziehungsweise bearbeitet. Ziel ist es, eine relativ intuitive Bedienung zu ermöglichen, deren Logik sich an die gängiger Grafikprogramme anlehnt. Auf der anderen Seite haben wir viel Wert auf ein ständiges Benutzerfeedback gelegt, so dass viele fehlerhafte Konstruktionen gar nicht erst möglich sind.

Der Editor führt, gesteuert durch die grafischen Eingaben des Benutzers, ständig ein grafisches und logisches Modell des entsprechenden Statecharts mit. Dieses Modell kann in mehreren Fenstern, in verschiedenen Teilansichten und verschiedenen Darstellungsgrößen und -tiefen gleichzeitig bearbeitet werden.

Neben dieser rein optischen Vergrößerung gibt es die Möglichkeit, Ansichten eines Zustandes zu öffnen. In dieser wird dann nur der entsprechende Zustand und seine Unterzustände betrachtet, sowie ein- und ausgehende Verbindungen. Wir haben uns

gegen die Möglichkeit einer partiellen Vergrößerung in einer Ansicht entschieden, da dies eine automatische Platzierung der Komponenten erfordert. Hingegen bieten wir mit der Möglichkeit, mehrere Fenster des gleichen Statecharts in unterschiedlichen Vergrößerungen zu öffnen, eine ähnliche Editiermöglichkeit, die aber immer die absolute Positionierung sicherstellt.

Um die grafische Ansicht übersichtlich zu halten und nicht zu überlasten, werden nichtgrafische Informationen in einem separaten Fenster, dem *Inspektor*, dargestellt. Hier werden alle Daten der markierten Objekte zum Editieren angezeigt. Änderungen wirken sich sofort auf alle Ansichten aus.

Der Dialog-Editor

Neben dem oben erwähnten Statechart-Editor bieten wir einen einfachen Dialog-Editor, der Dialogkomponenten in einer Liste verwaltet. Auch die Eigenschaften der Dialoge werden im Inspektor editiert.

Eingabemasken

Zur Eingabe der Bedingungen, Aktionen und Ereignisse existieren Eingabemasken.

6.5 Der Simulator

Der Simulator dient der Simulation des erstellten Statecharts. Das Verhalten wird über die Zeit ermittelt und dargestellt. Benutzereingaben über Dialogkomponenten beeinflussen den Ablauf, der grafisch in allen offenen Ansichten dargestellt wird.

6.5.1 Verwaltete Daten

Der Simulator hat Zugriff auf die Daten, die für den Ablauf der Simulation notwendig sind. Dazu gehören insbesondere:

- der Konfigurationsvektor mit den aktiven Zuständen
- eine Ereignismenge, welche die momentan sichtbaren Ereignisse enthält
- eine Aktionsmenge, welche die auszuführenden Aktionen enthält
- den Zeitzähler
- die History-Hierarchie für jeden Zustand, bei dem ein History-Einstieg möglich ist

Diese Daten teilen sich Simulator und Simulationsmodell (SimSCM). So werden Zustandsvektor und History-Hierarchie im Simulationsmodell gespeichert. In einem Zeitschritt führt der Simulator dann die entsprechenden Transitionen durch. Die Simulationsobjekte führen selbst die nebenläufigen Aktivierungen (wie Aktivieren von Ober- und Unterzuständen) durch und liefern dem Simulator die dabei erzeugten Aktionen zurück; dieses Verfahren wird in Abschnitt 6.5.3 noch genauer beleuchtet. Der Simulator hält die Ereignis- und Aktionsmenge, welche auf diesem Weg aktualisiert

beziehungsweise in den entsprechenden Phasen ausgeführt werden. Der Zeitzähler wird auch vom Simulator aktualisiert und dient ausschließlich der Benutzerinformation.

Der Simulator benötigt Zugriff auf beziehungsweise Interaktion mit folgenden Komponenten:

- den Dialogelementen,
- dem logischen Statechart-Modell,
- dem Simulationsmodell,
- den Ansichten.

Zu Beginn der Simulation erzeugt der Simulator aus dem logischen Dialogmodell (LogDM) die Dialoge. Von diesen empfängt er während der Simulation Aktionen und kann seinerseits durch Aktionen aus dem Statechart mit ihm interagieren. Zugriff auf das logische Statechart-Modell und dem Simulationsmodell benötigt der Simulator, um daraus die folgenden Transitionen zu berechnen. Die berechneten Übergänge werden dann dem Simulationsmodell mitgeteilt, welches diese ausführt und die dabei generierten Aktionen zurückliefert. Anschließend werden dann vom Simulator die Ansichten aktualisiert.

Einen Überblick über die Hierarchie der für die Simulation benötigten Klassen gibt Abbildung 6.7.

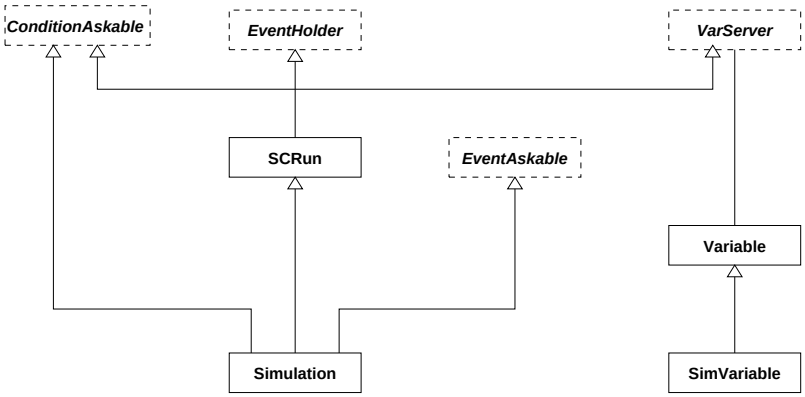


Abbildung 6.7: Hierarchie der Simulation

6.5.2 Der Simulationsablauf

Der Ablauf einer Simulation besteht aus den folgenden fünf Phasen, die von der Klasse *SCRun* gesteuert werden:

Initialisierung. Es wird ein Simulationsmodell erzeugt und die Objekte werden mit ihren Startwerten versehen.

- Die Aktionsmenge und die Ereignismenge werden geleert.

- Die Dialoge werden erzeugt.
- Der Zeitzähler wird auf 0 gesetzt.
- Die Historyhierarchie wird geleert.

Externe Aktionen sammeln. Es werden externe Aktionen vom Benutzer gesammelt, wie zum Beispiel das Drücken von Buttons. Die dabei generierten Aktionen werden zur Aktionsmenge hinzugefügt.

Diese Phase wird beendet, wenn der Zeitschritt abgelaufen ist (durch eine Uhr oder durch einen manuellen Schritt-Befehl).

Externe Aktionen auswerten.

- Die in der vorherigen Phase gesammelten Aktionen werden ausgeführt und die dabei generierten Ereignisse zur Ereignismenge hinzugefügt.
- Die Aktionsmenge wird geleert.

Übergänge berechnen Aus der Ereignismenge, dem logischen Statechartmodell und dem Simulationsmodell werden die Transitionen ermittelt.

Übergänge ausführen

- Die berechneten Übergänge werden dem Simulationsmodell zum Setzen übergeben. Zur Aktionsmenge werden die dabei generierten Aktionen hinzugefügt.
- Die Historyhierarchie wird implizit vom Simulationsmodell angepasst.

Übergänge abschliessen

- Die Zeit wird um eine Einheit erhöht.
- Die Ereignismenge wird geleert.
- Die Ansichten werden aktualisiert.

Interne Aktionen auswerten

- Die in der Phase **Übergange ausführen** gesammelten Aktionen werden abgearbeitet und dabei generierten Ereignisse zur Ereignismenge hinzugefügt.
- Die Aktionsmenge wird geleert.

↻ Fahre fort mit **Externe Aktionen sammeln**.

Dieser Ablauf wird in Abbildung 6.8 dargestellt. Dabei stellt die mittlere Achse die einzelnen Schritte dar, die solange wiederholt werden, bis der Benutzer die Simulation abbricht oder ein Endzustand aktiviert wird. Auf der rechten Seite werden die im Simulator verwalteten Objekte angezeigt; die Verbindungen zum Ablauf stellen dar, wann und wie sie benutzt werden. Analoges gilt für die linke Seite der Abbildung mit den Objekten, die außerhalb des Simulators gehalten werden.

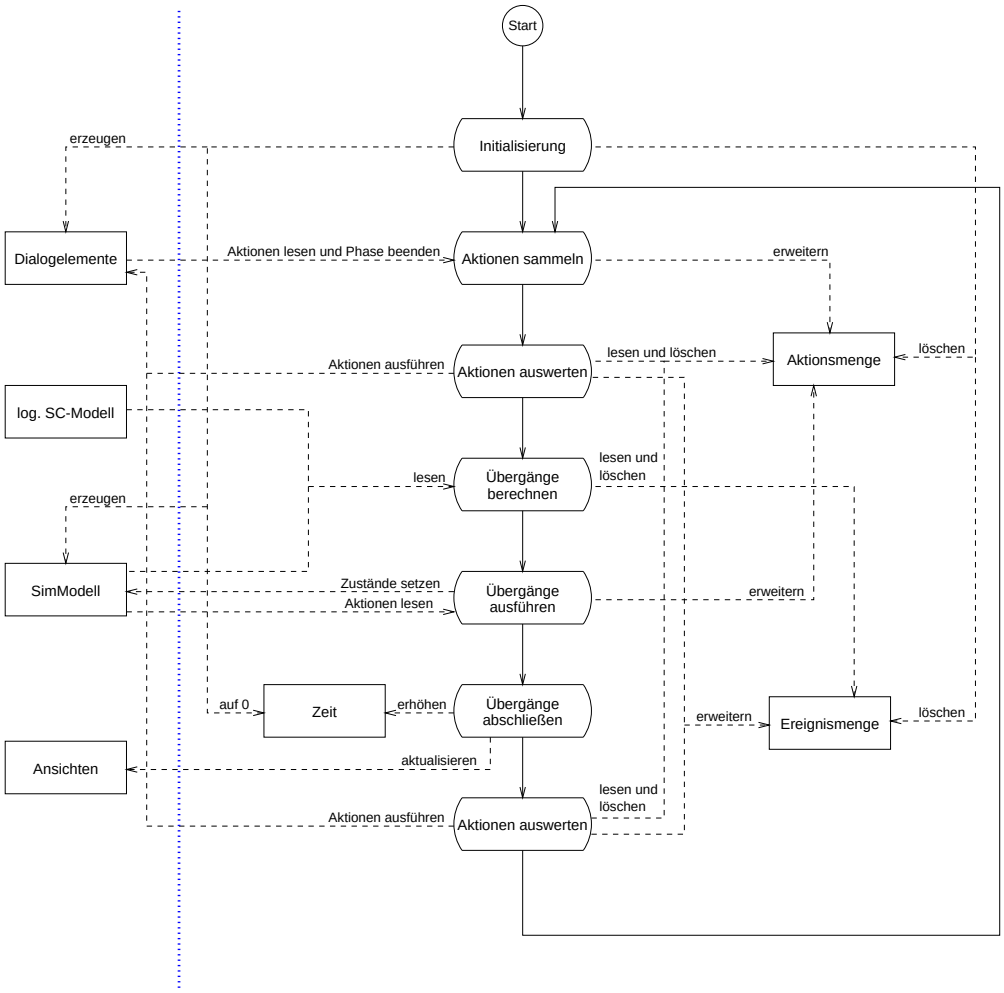


Abbildung 6.8: Übersicht des Simulationsablaufs

6.5.3 Aktivieren in der Zustandshierarchie

Wie Abschnitt 6.2.1 erwähnt, verwaltet das Simulationsmodell die Information, welche Zustände aktiviert sind, durch eine objektorientierte Zustandshierarchie.

Das Deaktivieren von Zuständen in dieser Hierarchie ist unproblematisch, da sämtliche Unterzustände deaktiviert werden müssen. Welche Oberzustände dabei ebenfalls deaktiviert werden sollen, kann nicht ausschließlich in der Hierarchie festgestellt werden, da diese Information von der Lage des ausgeführten Übergangs abhängt. Deshalb übernimmt der Simulator[4] selbst die Auswahl des obersten zu deaktivierenden Zustands.

Wenn ein Zustand aktiviert wird, so müssen auch sämtliche Oberzustände aktiviert werden, solange sie bisher deaktiviert sind. Außerdem müssen ausgewählte Unterzustände durch ihre Default-Einstiege (oder gegebenenfalls History-Einstiege)

[4] diese Aufgabe wird wieder von der Klasse SCRun übernommen, die ja als Oberklasse der Simulation und des als Quelltext exportierten Statecharts auftritt

aktiviert werden. Diese Informationen liegen sämtlich im Simulationsmodell, das diese zusätzlichen Aktivierungen deshalb selbständig durchführt. Dabei wird wie folgt vorgegangen.

Zunächst werden die atomaren Zustände festgestellt, die aktiviert werden müssen. Dies geschieht entweder über Default-Einstiege oder, wenn ein Deep-History-Einstieg benutzt wurde, in jeder Ebene der Hierarchie durch die History der Zustände. Bei AND-Zuständen wird dieses Aktivieren in jeder Partition durchgeführt. Dabei wird in der Zustandshierarchie solange hinabgestiegen, bis ein atomarer Zustand angetroffen wird.

Von diesem atomaren Zustand aus werden nun alle Oberzustände aktiviert,[5] bis ein bereits aktivierter Oberzustand angetroffen wird. Aktionen, die bei diesem Aktivieren ausgelöst werden, werden während dieses Aufsteigens in der Hierarchie zentral gespeichert und später vom Simulator übernommen.

Problematisch sind Übergänge, die in mehreren Partitionen eines AND-Zustands enden. Da die Übergänge vom Simulator und nicht im Simulationsmodell verwaltet werden, wird dem Modell für jeden Endzustand separat mitgeteilt, dass dieser aktiviert werden muss. Dabei dürfen in den anderen Partitionen selbstverständlich nicht automatisch die Default-Einstiege aktiviert werden. Andererseits wird von der Syntax der Statecharts nicht verlangt, dass in diesen Fällen in jeder Partition ein Endzustand des Übergangs liegen muss. Daher kann es vorkommen, dass nach Aktivieren aller Endzustände einige Partitionen immer noch nicht aktiv sind. Um dies zu verhindern, werden die AND-Zustände nach Durchführen aller Übergänge „nachgeführt“. Das heißt, es wird bei allen AND-Zuständen, die in diesem Zeitschritt aktiviert wurden, überprüft, ob alle Partitionen aktiviert sind. Partitionen, bei denen dies nicht der Fall ist, werden dann mittels ihres Default-Einstiegs aktiviert, so dass das Statechart konsistent ist.

Beispiel.

Abbildung 6.9 soll dieses Verfahren an einem Beispiel erläutern. Zu Anfang sei Zustand B aktiv, und damit auch Zustand A. Nun soll der Übergang t_1 durchgeführt werden, der zwei Endzustände besitzt. Dazu wird zunächst B deaktivert; Zustand A bleibt allerdings aktiviert, da er von t_1 nicht verlassen wird. Der Simulator wird dem Simulationsmodell nun mitteilen, dass Zustand E zu aktivieren ist. Da dies bereits ein atomarer Zustand ist, werden lediglich die Oberzustände aktiviert, das heißt die Zustände E, D und C; da A bereits aktiv ist, wird die Hierarchie nicht weiter nach oben durchgegangen (Zustand A kann ja durchaus noch weitere Oberzustände besitzen). Als nächstes teilt der Simulator dem Modell mit, dass Zustand G aktiviert werden soll. Dabei wird nun in die Hierarchie hinabgestiegen, bis der atomare Zustand H erreicht ist. Von diesem aus wird die Hierarchie aufsteigend durchlaufen, das heißt die Zustände H und G werden aktiviert. Da C bereits aktiviert ist, bricht diese aufsteigende Aktivierung bereits früher ab als die in Partition *P1* gestartete.

Angenommen, damit sind alle Übergänge des Zeitschrittes geschehen. Nun werden alle gerade aktivierten AND-Zustände überprüft, ob sie nachgeführt zu werden brauchen. Das ist bei Zustand C der Fall, da die Partition *P3* immer noch keinen aktiven Zustand enthält. Nun wird hier also der Default-Einstieg benutzt und damit Zustand J aktiviert. Wie beim üblichen Aktivieren wird auch hier zunächst in die Hierarchie hinabgestiegen, und beim anschließenden aufsteigenden Durchlauf die jeweiligen

[5] genauer gesagt werden die Zustände als „im nächsten Zeitschritt aktiv“ markiert

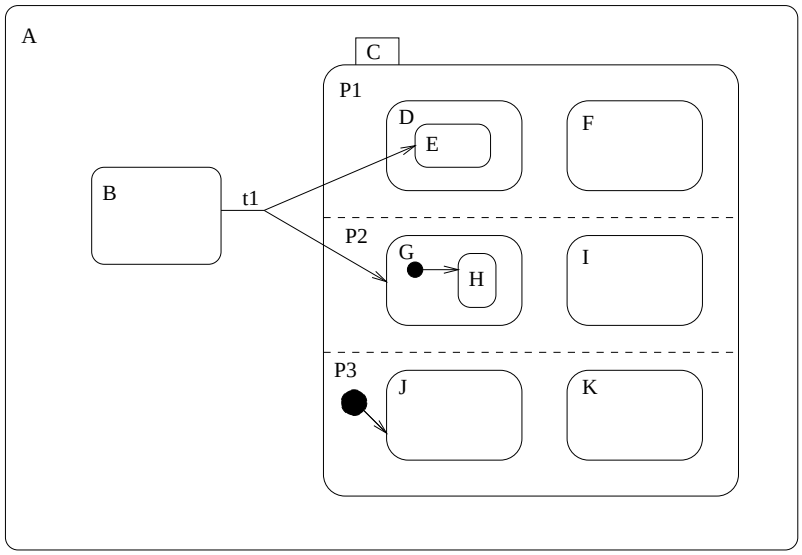


Abbildung 6.9: Ablauf der Aktivierung im Simulationsmodell

Zustände aktiviert, was sich bei diesem Beispiel jedoch auf den Zustand `J` beschränkt. Im Statechart sind nun also die Zustände `A, C, D, E, G, H` und `J` aktiviert, und es kann mit der Simulation fortgefahren werden.

6.6 Der Verifikator

Zur Funktionsweise des Verifikators muss nicht viel erläutert werden, da er hauptsächlich aus einzelnen Test-Routinen besteht. Die Auswahl des Benutzers, welche Tests durchgeführt werden sollen, wird in der Klasse *VProperties* gespeichert, die dem Verifikator übergeben wird. Die Ausgabe erfolgt im *VWindow*, in dem auch den einzelnen Warnungen und Fehlermeldungen die entsprechenden dargestellten Objekte zugeordnet sind.

6.7 Quelltextexport

Die Aufgabe des Quelltextexports ist der Simulation sehr ähnlich. Prinzipiell soll das, was während der Simulation geschieht, in einem eigenständigen Programm durchgeführt werden. Von diesem Ansatz her haben wir den Quelltextexport derart gestaltet, dass jede Komponente ihren Code in eine Datei schreiben kann. Dabei erzeugt beispielsweise die Dialogkomponente Button den Code, der zum erzeugen eines *javax.swing.JButton* nötig ist. Die erzeugten Aktionen werden durch *java.awt.event.ActionListener* empfangen. Das Statechart selbst erzeugt ebenfalls seinen Aufbau innerhalb einer Datei, und eine modifizierte Simulations-Klasse erzeugt eine analoges Modell. Letztendlich konnten wir so prinzipiell die gleichen Klassen benutzen, wie sie auch zur Simulation verwendet werden. Der Aufbau im Überblick zeigt Abbildung 6.10.

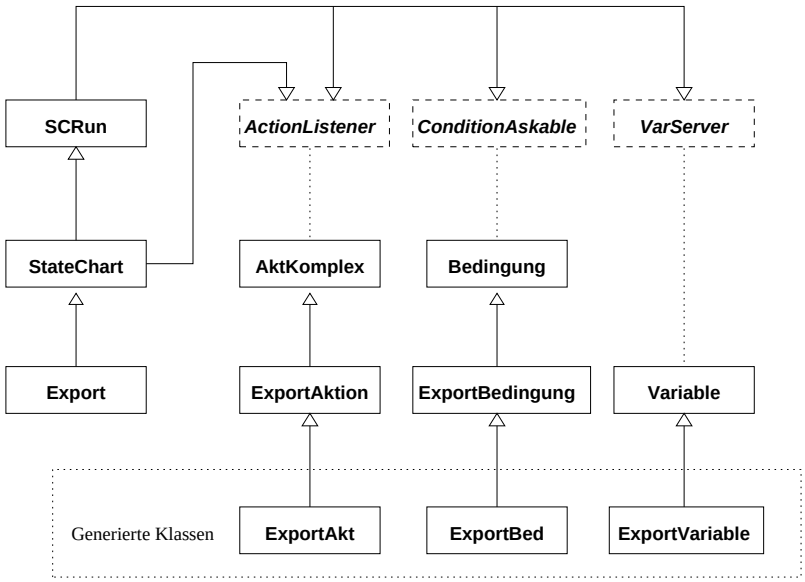


Abbildung 6.10: Hierarchie des Quelltextexports

6.8 Das Kommandointerface – Laden und Speichern

Wir haben das Programm mit einem Kommandointerface ausgestattet, so dass der Benutzer die Möglichkeit hat, die meisten Editiervorgänge durch Textkommandos einzugeben. Der Vorteil dieser einheitlichen Schnittstelle für alle Funktionen liegt in der einfachen Möglichkeit, Automatismen einzuführen. Weiterhin haben wir das Laden und Speichern in diesem Format implementiert, so dass die gespeicherten Dateien in einem Klartextformat vorliegen und so auch manuell noch geändert werden können. Die Sprache, in der wir die Kommandos erlauben, ist recht einfach aufgebaut, um das Parsen möglichst schnell durchzuführen. So wird nur ein Kommando je Zeile erlaubt. Eine Zeile wird nach erfolgreichem Parsen eines Befehls nicht weiter geparst. Es wird nicht zwischen Groß- und Kleinschreibung unterschieden.

Die Tabelle C ab Seite 175 enthält die Grammatik der Sprache.

In diesem Zusammenhang ist auch die Implementation der Undo-Funktionalität zu erwähnen. *SCed* speichert die im Editor vorgenommenen Änderungen auf einem Stack, indem hier das Statechart in der eben erwähnten Sprache abgelegt wird. Wird eine Änderung zurückgenommen, so wird das aktuelle Statechart auf einem Redo-Stack abgelegt, so dass sowohl weitere Undo- als auch Redo-Aufrufe möglich sind.

6.9 Ausblick

Es gibt selbstverständlich viele Möglichkeiten, wie das Programm weitergeführt werden könnte. Die wohl strengste Einschränkung, die *SCed* den Statecharts auferlegt, ist die Festlegung auf einen einzigen Namensraum. Eine naheliegende Erweiterung des Programms wäre es daher, mehrere Statecharts zuzulassen. Dadurch könnten getrennte Namensräume verwendet werden, durch die eine echte AND-Parametrisierung

möglich wird. Auch die starke Rahmenbedingung ließe sich dadurch in naheliegender Weise umsetzen. Die Umstellungen, die hierfür notwendig werden, sind jedoch sehr umfangreich.

Einige kleinere Modifikationen bestünden etwa darin, weitere der im Pflichtenheft in Abschnitt B.1.2 aufgeführten Kann-Kriterien umzusetzen, wie etwa die statischen Reaktionen[6] oder die Diagramm-Konnektoren.[7] Auch das Hinzufügen von verzögerten Aktionen[8] wäre ohne großen Aufwand machbar.[9] Andere Punkte der Kann-Kriterien würden einen deutlichen Umbau des Programms erfordern, wie etwa das Umsetzen eines alternativen Zeitmodells[10] oder einer objektbezogenen Undo/Redo-Funktion.[11] Darüberhinaus könnte auch eine Möglichkeit vorgesehen werden, externe Ereignisse als Quelltext einzugeben; es müsste hierbei jedoch gegebenenfalls die Rolle der Ereignisse im Quelltextexport umgestellt werden.[12] Bedingungen an Default-Einstiegen haben wir allerdings ausgeschlossen, obwohl diese Möglichkeit in den Kann-Kriterien aufgeführt wurde, weil sie die Übersichtlichkeit des Statecharts stören.[13] Bei den Bedingungen ließe sich zusätzlich eine `else`-Bedingung einführen.

Als Erweiterung des Dialog-Editors wäre denkbar, dem Benutzer die Möglicheit zu geben, auch das grafische Layout des erzeugten Dialogs zu bearbeiten. Das Problem hierbei liegt allerdings nicht in der Struktur von *SCed*, wo sich entsprechend erweiterte Dialogkomponenten relativ einfach einfügen ließen, sondern im Umfang, den ein Dialog-Editor erfordert, um zu einem komfortablen Werkzeug zu werden.

Das Programm ließe sich auch weiterführen, indem als zusätzliches Werkzeug noch ein Programmstruktur-Editor hinzugefügt wird, wie er in Abschnitt 3.1.3 beschrieben wurde.

[6] vgl. die Beschreibung auf Seite 39

[7] vgl. Abschnitt 31

[8] vgl. Abschnitt 2.7

[9] Im Gegensatz dazu ließe sich die Implementation von aufgeschobenen Ereignisse im Sinne von Abschnitt 2.5.1 nicht so einfach ergänzen, da dies tiefer in die Konzeption der Ereignisse eingreift.

[10] vgl. Abschnitt 2.8

[11] vgl. hierzu die Überlegungen auf Seite 63

[12] vgl. die Gegenüberstellung der zwei Möglichkeiten auf Seite 87.

[13] vgl. die Argumentation in Abschnitt 4.2.1

Teil III

Anhang: Beispiel und Programmdokumentation

They look into the beauty of thy mind,
and that, in guess, they measure by thy deeds.
(Shakespeare, Sonnet 69)

Anhang A

Beispiele

A.1 Die Teemaschine

Statecharts eignen sich nicht nur dafür, Software und Benutzerschnittstellen zu modellieren. Sie lassen sich auch auf andere interaktive Systeme anwenden, wie das folgende Beispiel einer Teemaschine zeigen soll.

A.1.1 Beschreibung des Systems

Zunächst soll der Aufbau und die Benutzung der Teemaschine (Abbildung A.1) beschrieben werden, um daraus im zweiten Schritt die Struktur des Statecharts ableiten zu können.

Die Teemaschine besitzt zwei Kontrollelemente, nämlich einen Ein/Ausschalter und einen Drehknopf, an dem eingestellt werden kann, wie lange der Tee ziehen soll. Außerdem gibt es mehrere Behälter, in denen sich das Wasser zu unterschiedlichen Zeitpunkten der Teezubereitung befindet. Anfangs wird es in einen Wassertank eingefüllt, in dem es erhitzt wird. Ist dies geschehen, fließt das Wasser in den Ziehbehälter, in den vorher bereits Tee eingefüllt wurde. In diesem Moment wird eine Uhr gestartet, um die eingestellte Ziehzeit zu messen. Am Ende dieser Zeitdauer wird das Wasser in die Teekanne geleitet, und der Tee ist fertig.

A.1.2 Elemente des Dialogs

Um die Teemaschine durch *SCed* simulieren zu können, müssen die Zugriffsmöglichkeiten durch Dialogelemente beschrieben werden. Dies wird durch die folgenden Elemente realisiert (siehe Abbildung A.2):

Strom eingeschaltet Diese Checkbox gibt an, wie der Ein/Ausschalter momentan steht.

Ziehzeit In diesem Textfeld kann die Zeit eingegeben werden, die der Tee ziehen soll. Dieses Dialogelement entspricht dem Drehregler der echten Teemaschine.

Abbildung A.1: Die Teemaschine, die simuliert wird

Ziehzeittimer Die Stoppuhr für die Ziehzeit erscheint nur während der Simulation; im exportierten Quelltext wird die Zeit nur intern gemessen.[1]

Wasser einfüllen Durch betätigen dieses Knopfes wird der Wassertank gefüllt.[2]

mit Tee gefüllt Mit dieser Checkbox wird angegeben, ob sich momentan Tee im Ziehbehälter befindet.

Teekanne leeren Soll die Teekanne geleert werden, so geschieht dies durch Betätigen dieses Knopfes.

Wasser ist heiß Das Ereignis, dass das Wasser im Tank nun kocht, wird in der Simulation vom Benutzer durch Drücken dieses Knopfes eingegeben. Im erzeugten Quelltext ist dieser Knopf nicht mehr sichtbar.[3]

Darüber hinaus gibt es noch ein Popup, das geöffnet wird, wenn sich die Teekanne füllt. In diesem Fenster wird dem Benutzer angezeigt, ob sie nun mit Wasser, Tee oder

[1] vgl. Abbildung A.2, in der die Dialoge der Simulation und des exportierten Programms dargestellt sind.

[2] Um das Modell einfach zu halten, ist es nicht möglich, den Tank beispielsweise nur zur Hälfte zu füllen; er ist entweder ganz voll oder leer.

[3] vgl. wieder Abbildung A.2.

Abbildung A.2: Die Dialoge der Teemaschine

Brühe gefüllt ist. Das Popup besitzt nur einen einzigen Knopf, der mit „OK“ beschriftet ist.

A.1.3 Umsetzung in ein Statechart

Die globale Struktur

Es fällt auf, dass die Teemaschine recht viele parallele Teile besitzt. So ist beispielsweise die Angabe, wo sich das Wasser gerade befindet (oder ob bereits wieder neues Wasser eingefüllt wurde, obwohl der Tee gerade zieht), nicht davon abhängig, ob der Strom eingeschaltet ist. Daher ist die Unterteilung des globalen Statecharts in mehrere Partitionen naheliegend, die jeweils den Zustand eines dieser Elemente darstellen. Wir haben diese Einteilung wie folgt vorgenommen:

Strom Hier wird angezeigt, in welcher Stellung sich der Ein/Ausschalter befindet.

Ziehzeittimer Diese Partition stellt die Stoppuhr für die Ziehzeit dar und gibt an, ob diese Uhr gerade läuft oder nicht.

Wasserbehälter Wenn Wasser in den Tank eingefüllt wurde, wird das durch diese Partition modelliert.

Tee im Ziehbehälter Die Angabe, ob sich gerade Tee im Ziehbehälter befindet, und ob dieser frisch ist oder bereits zur Zubereitung einer Kanne Tee benutzt wurde, ist in dieser Partition angegeben.

Wasser im Ziehbehälter Ist das Wasser erhitzt und in den Ziehbehälter geflossen, so ist das in dieser Partition zu erkennen.

Teekanne Es gibt mehrere Möglichkeiten, wie die Teekanne gefüllt sein kann; diese werden in der letzten Partition modelliert.

Wird die Teemaschine auf diese Weise durch ein Statechart beschrieben, so wird dieses etwa wie in Abbildung A.3 aussehen. In dieser Darstellung sind die Pfeile nicht beschriftet, um das Statechart übersichtlicher zu halten. Auf genauere Details der einzelnen Partitionen und die Zusammenhänge zwischen ihnen wird in den folgenden Abschnitten genauer eingegangen.

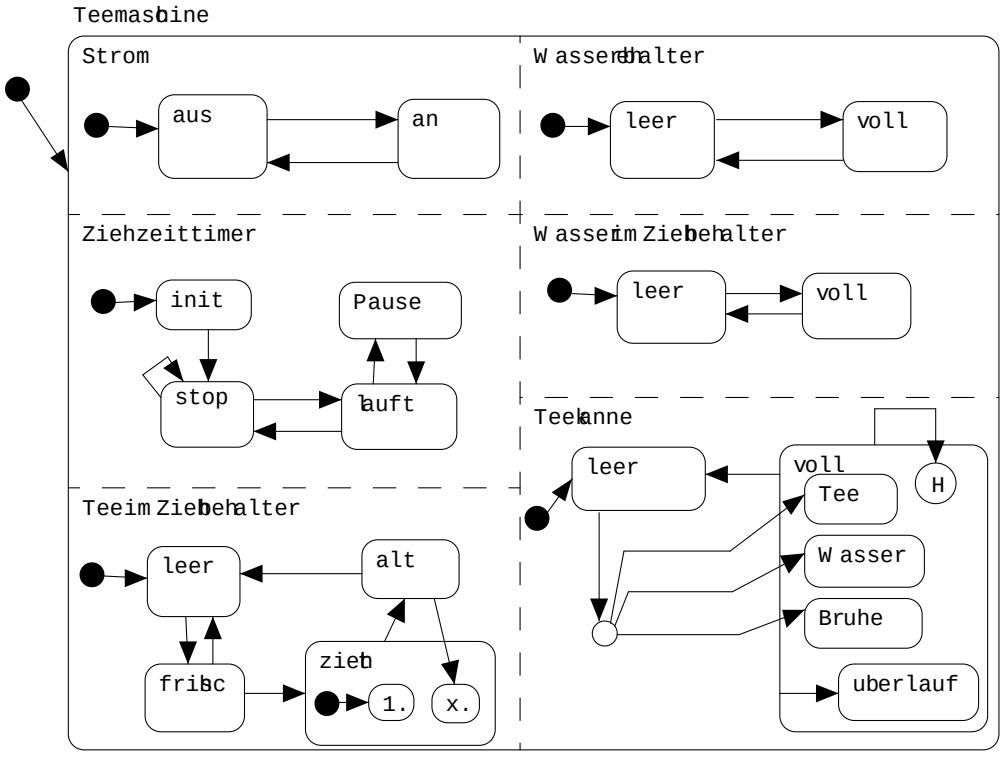


Abbildung A.3: Die Gesamtansicht der Teemaschine

Liste der Ereignisse und Aktionen

Der Übersichtlichkeit halber werden die Ereignisse und Aktionen in den hier dargestellten Statecharts nur abgekürzt angegeben. Zum besseren Verständnis folgen deshalb zunächst zwei Listen sämtlicher Ereignisse und Aktionen, die in diesem Modell auftreten.

- Die folgenden Ereignisse werden vom Benutzer beim Bedienen der Teemaschine ausgelöst:

  EA der Ein/Ausschalter wurde gedrückt

  ZZ der Wert der Ziehzeit wurde geändert

  WIn Wasser wurde in den Tank eingefüllt

  Tee Tee wurde eingefüllt oder entfernt

  TkLeer die Teekanne wurde geleert

Tk im Popup, das den Zustand der Teekanne angibt, wurde auf den OK-Knopf gedrückt

- Ein Ereignis wird prinzipiell von der Teemaschine selbst ausgelöst; in der Simulation muss es jedoch der Benutzer über den Systemdialog eingeben:[4]

  Wih das Wasser ist heiß

- Das Statechart löst während des Ablaufs die folgenden Ereignisse in der hier genannten Reihenfolge aus:

  WzZ das Wasser fließt in den Ziehbehälter

  TI die Stoppuhr der Ziehzeit ist abgelaufen

  WzTk das Wasser fließt in die Teekanne

Die Übergänge, sowie das Betreten oder Verlassen von Zuständen können die in der folgenden Liste genannten Aktionen aufrufen. In den Zuständen sind diejenigen Aktionen, die ausgelöst werden, wenn der Zustand betreten wird, durch die Angabe `enter` gekennzeichnet, und solche, die beim Verlassen ausgelöst werden, durch `exit`.

- Der Benutzerdialog wird über die folgenden Aktionen beeinflusst:

  visible!(ZZ, *boolean*) das Eingabefeld für die Ziehzeit sichtbar beziehungsweise unsichtbar machen

  visible!(Tee, *boolean*) die Checkbox, die angibt, ob Tee eingefüllt ist oder nicht, sichtbar beziehungsweise unsichtbar machen

  visible!(Tk, *boolean*) das Popup, in dem der Zustand der Teekanne angegeben wird, anzeigen oder schließen

- Im Zusammenhang mit dem Timer für die Ziehzeit werden die folgenden Aktionen benutzt:

  set!(ZZ=15) den Wert der Ziehzeit auf 15 Sekunden setzen

  set!(TI=ZZ) die Uhr mit dem Wert der Ziehzeit initialisieren

  on!(TI) die Uhr starten

  off!(TI) die Uhr anhalten

  TI das Ereignis „Stoppuhr abgelaufen“ auslösen

- Wenn bestimmte Phasen der Teezubereitung abgeschlossen sind, fließt das Wasser in andere Behälter der Teemaschine. Dies wird durch die folgenden zwei Aktionen dargestellt:

  WzZ das Ereignis „Wasser zum Ziehbehälter“ auslösen

  WzTk das Ereignis „Wasser zur Teekanne“ auslösen

[4] vgl. hierzu die Diskussion der externen Ereignisse in der Simulation auf Seite 80.

- Ist der Ziehvorgang abgeschlossen, fließt das Wasser in die Teekanne. Dem Benutzer wird mittels eines Popups angezeigt, was sich nun in der Kanne befindet. Wie in Abschnitt 76 auf Seite 69 beschrieben, wird der entsprechende Text über eine Variable gesetzt. Dies geschieht mit den folgenden Aktionen:

  set!(Tk="Tee") den Text für das Fenster der Teekanne auf „Die Kanne ist jetzt voll mit frischem Tee“ setzen

  set!(Tk="Wasser") den Text für das Fenster der Teekanne auf „Die Kanne ist jetzt voll mit heißem Wasser“ setzen

  set!(Tk="Brühe") den Text für das Fenster der Teekanne auf „In der Kanne ist jetzt eine ziemliche Brühe“ setzen

  set!(Tk="übergelaufen") den Text für das Fenster der Teekanne auf „Die Kanne ist gerade übergelaufen“ setzen

Partition 1: Strom

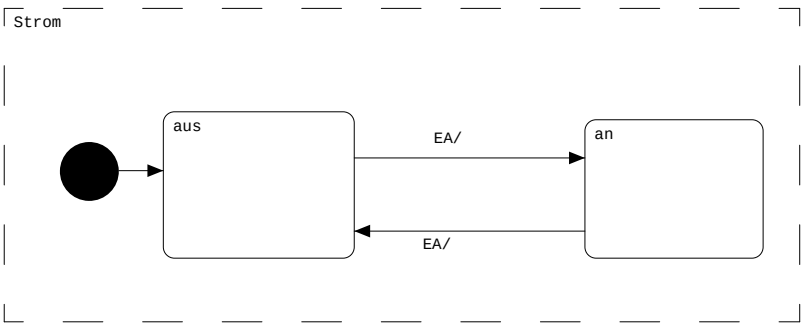


Abbildung A.4: Die Partition für den Ein/Ausschalter

Das Verhalten dieser Partition ist denkbar einfach, denn sie modelliert hauptsächlich die Checkbox „Strom eingeschaltet“.

Beim Start des Systems ist die Teemaschine ausgeschaltet. Dies bleibt solange der Fall, bis der Ein/Ausschalter gedrückt wird (Ereignis *EA*). Das System geht dann in den Zustand `an` über. Wird der Ein/Ausschalter erneut betätigt, ist der Strom wieder ausgeschaltet.

Partiton 2: Ziehzeittimer

Beim Start des Systems wird zunächst das Textfeld, das die Ziehzeit angibt (`ZZ`), auf 15 Sekunden gesetzt. Im nächsten Zeitschritt geht das System in den Zustand `Stop` über, der angibt, dass die Uhr nicht läuft. Beim Betreten dieses Zustands wird der Timer `TI` mit dem Wert des Textfeldes initialisiert und das Textfeld `ZZ` sichtbar gemacht, damit der Wert umgestellt werden kann. Wird in diesem Textfeld ein neuer Wert eingegeben, wird dadurch das Ereignis *ZZ* ausgelöst. Dies führt dazu, dass der Zustand zunächst verlassen und sofort wieder betreten wird. Dadurch wird der Timer mit dem neuen Wert initialisiert.

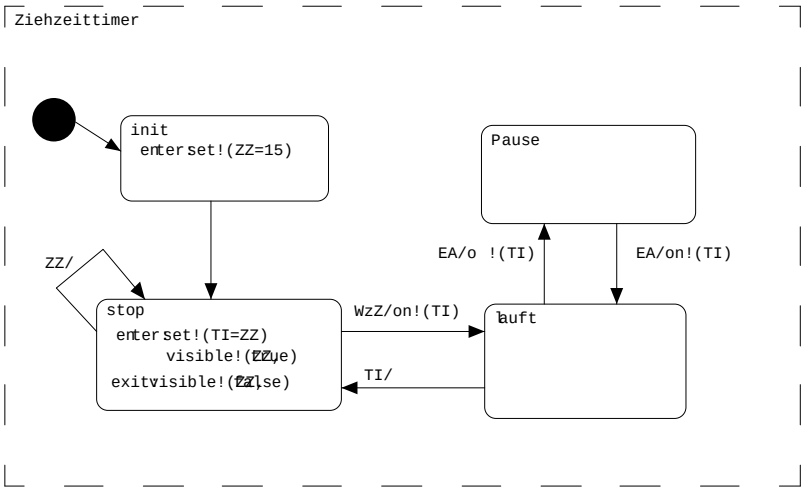


Abbildung A.5: Die Partition für die Stoppuhr

Wenn das Wasser in den Ziehbehälter fließt, wird das Ereignis *WzZ* ausgelöst und dadurch der Zustand `stop` verlassen. Dies führt dazu, dass der Wert im Textfeld der Ziehzeit nicht mehr verändert werden kann und der Timer `TI` gestartet wird. Das System befindet sich dann solange im Zustand `läuft`, bis entweder der Ein/Ausschalter betätigt wurde, oder der Timer abgelaufen ist. Im ersten Fall wird der Timer angehalten und das System geht in den Zustand `Pause` über, in dem es so lange bleibt, bis der Ein/Ausschalter wieder gedrückt wird. Im zweiten Fall wird das Ereignis *TI* von der Uhr ausgelöst, und das System springt in den Zustand `stop` zurück, wodurch die Uhr wieder auf den Wert des Textfeldes zurückgesetzt wird.

Partiton 3: Wasserbehälter

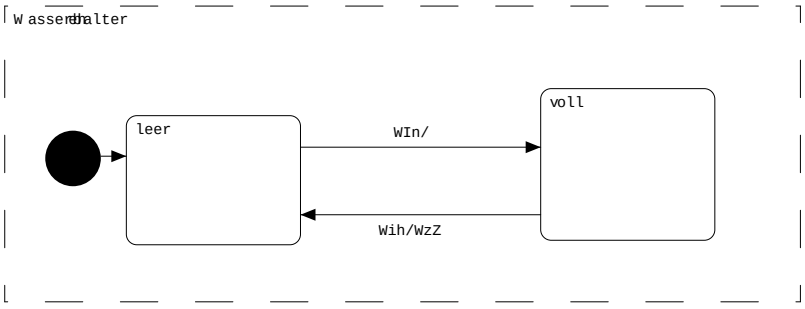


Abbildung A.6: Die Partition für den Wassertank

Diese Partition ist ähnlich einfach wie Partition 1. Allerdings wird hier keine Checkbox modelliert, da der Übergang von `voll` auf `leer` nicht vom Benutzer sondern vom System ausgelöst wird.

Der Wasserbehälter ist zunächst leer. Wird Wasser eingefüllt (*WIn*), so geht das System in den Zustand `voll` über. Wird das Ereignis *Wih* ausgelöst, so geht das System

wieder in den Zustand `leer` über und lässt das Wasser in den Ziehbehälter fließen (Ereignis *WzZ*).

Partiton 4: Tee im Ziehbehälter

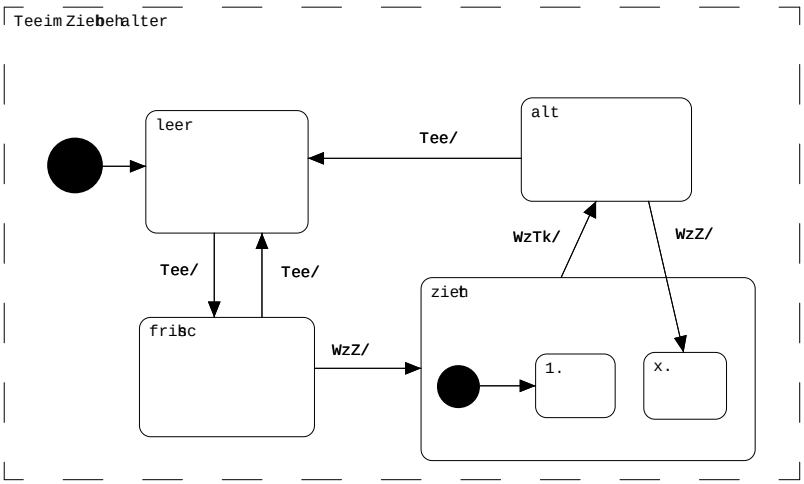


Abbildung A.7: Die Partition, die den Tee im Ziehbehälter beschreibt

Am Anfang ist noch kein Tee in den Ziehbehälter eingefüllt. Wird das Ereignis *Tee* ausgelöst, das anzeigt, dass Tee eingefüllt oder entfernt wurde, so wechselt das System entsprechend zwischen den Zuständen `leer` und `frisch`. Ist der Zustand `leer` aktiviert, wenn das Ereignis *WzZ* eintritt, so bleibt dieser erhalten. Vom Zustand `frisch` aus wird in diesem Moment jedoch der Zustand `zieht` betreten. Dabei wird dessen Default-Einstieg benutzt, der auf den Zustand `1.` verweist. Dies gibt an, dass der eingefüllte Tee zum ersten Mal benutzt wird.

Beim Eintreten des Ereignisses „Wasser zur Teekanne“ (*WzTk*) wird der Zustand `zieht` wieder verlassen und der Zustand `alt` betreten. Dadurch wird angezeigt, dass der Tee im Ziehbehälter nicht mehr frisch ist. Wird das Ereignis *WzZ* ausgelöst, während dieser Zustand aktiv ist, wird der Zustand `x.` betreten, der ein Unterzustand von `zieht` ist: der Tee wird nicht zum ersten Mal verwendet. Auch dieser Zustand wird durch das Ereignis *WzTk* verlassen.

Wird im Zustand `alt` das Ereignis *Tee* ausgelöst, ist damit der Tee wieder entfernt, und das System geht in den Zustand `leer` über. Solange sich Wasser im Ziehbehälter befindet, kann Tee weder eingefüllt noch entfernt werden; das heißt, das Ereignis *Tee* kann nicht ausgelöst werden. Dies wird in Partition 5 dadurch gelöst, dass die entsprechende Checkbox des Benutzerdialogs unsichtbar gemacht wird.

Partiton 5: Wasser im Ziehbehälter

Zunächst ist kein Wasser im Ziehbehälter. Wenn jedoch das Ereignis „Wasser zum Ziehbehälter“ (*WzZ*) ausgelöst wird, wird er durch das Wasser aus dem Tank gefüllt. In diesem Moment wird die Stoppuhr für die Ziehzeit gestartet. Da der Ziehbehälter nun voll ist, kann momentan kein Tee entfernt oder eingefüllt werden; daher wird die

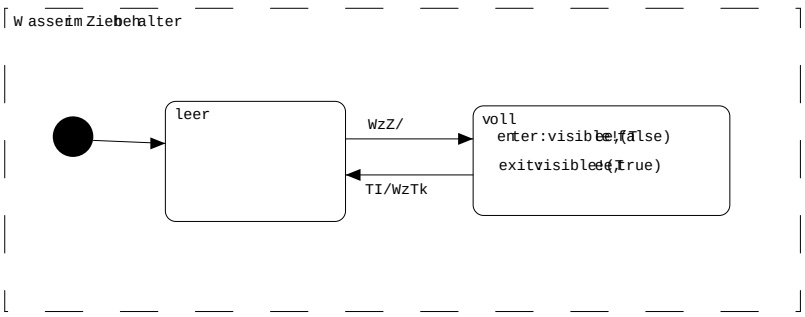


Abbildung A.8: Die Partition, die das Wasser im Ziehbehälter beschreibt

Checkbox bei Betreten des Zustands unsichtbar gemacht und erst bei Verlassen wieder aktiviert.

Ist die Stoppuhr abgelaufen (Ereignis *TI*), so geht das System wieder in den Zustand `leer` über, und das Wasser fließt in die Teekanne (Ereignis *WzTk*).

Partiton 6: Teekanne

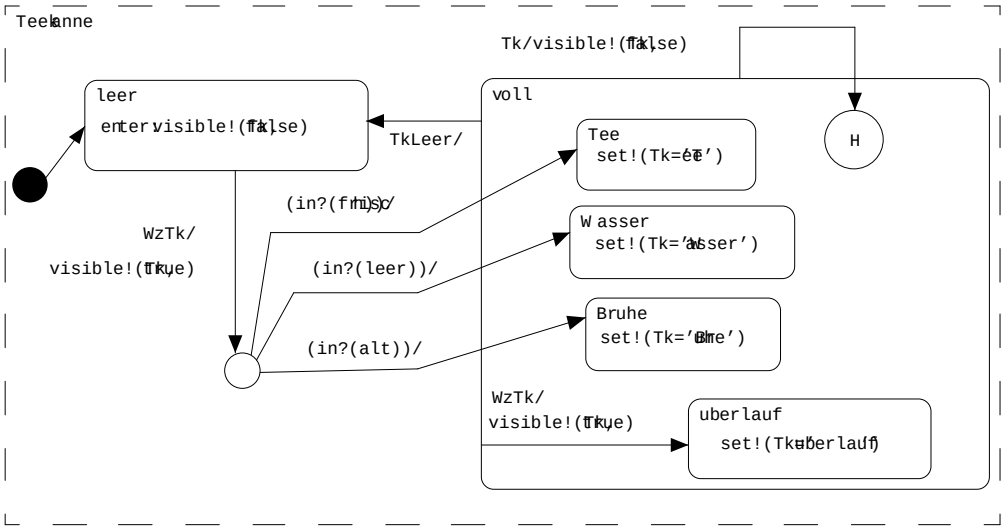


Abbildung A.9: Die Partition für die Teekanne

Beim Start der Maschine ist die Teekanne leer, und es wird keine Meldung angezeigt. Dies bleibt solange der Fall, bis Wasser in die Teekanne fließt (Ereignis *WzTk*). Je nach dem, ob sich während des Ziehens frischer oder alter Tee (oder gar keiner) im Ziehbehälter befand, ist die Kanne nun voll Tee, Brühe oder heißem Wasser.[5] Die Mel-

[5] Die angegebenen Bedingungen beziehen sich auf die Zustände in der Partition „Tee im Ziehbehälter“. Die scheinbare Mehrdeutigkeit der Angabe `leer` wird intern durch Benutzung von eindeutigen Identifizierern aufgehoben.

dung wird entsprechend gesetzt[6] und das Fenster angezeigt (`visible!(Tk, true)`). Wird im Zustand `voll` der Knopf des Fensters gedrückt (Ereignis *Tk*), so wird es wieder geschlossen; das System bleibt jedoch wegen des History-Einstiegs im selben Zustand. Durch Leeren der Teekanne (Ereignis *TkLeer*) kann das Fenster ebenfalls geschlossen werden; das System geht dann wieder in den Zustand `leer` über.

War die Kanne bereits gefüllt, wenn das Wasser aus dem Ziehbehälter abfließt, so läuft die Kanne über. Dies wird wieder durch eine entsprechende Meldung angezeigt. Da auch der Zustand `übergelaufen` ein Unterzustand von `voll` ist, kann das Fenster ebenfalls durch Drücken des OK-Knopfes geschlossen werden.

A.1.4 Unterschiede des Modells zur echten Teemaschine

Um das Modell einfach zu halten, gibt es für den Wassertank und das Teesieb nur jeweils zwei Zustände: voll oder leer. Es ist dadurch nicht möglich, den Tank beispielsweise nur zur Hälfte zu füllen; auch die Stärke des Tees kann (außer über die Ziehzeit) nicht variiert werden. Obwohl es mit der echten Teemaschine natürlich möglich ist, eine halbe Kanne sehr starken Tee durchlaufen zu lassen und den Rest dann mit heißem Wasser aufzufüllen, kann unser Modell diesen Vorgang nicht nachstellen. Mit der Hilfe von Variablen, die den Füllgrad der einzelnen Teile der Teemaschine angeben, ließe es sich entsprechend erweitern; wir haben uns jedoch dafür entschieden, in diesem Beispiel darauf zu verzichten.

A.2 Ein einfacher Anrufbeantworter

A.2.1 Beschreibung des Systems

Dieses Beispiel skizziert die grundlegende Verhaltensweise eines einfachen Anrufbeantworters. Um das Modell übersichtlich zu halten, wurde auf die tatsächliche Speicherung von Ansagetexten und Nachrichten verzichtet.

Wenn der Anrufbeantworter eingeschaltet ist, so wartet er zunächst auf einen neuen Anruf oder die Eingabe des Benutzers, die bisher aufgezeichneten Nachrichten abzuspielen oder zu löschen.

Bei einem eingehenden Anruf wird, solange noch Speicherplatz frei ist, zunächst eine Ansage abgespielt, und dann eine Nachricht des Anrufers aufgenommen. Ist der Speicher jedoch bereits voll, wird lediglich eine Ansage abgespielt. Jeder dieser Vorgänge kann entweder dadurch abgebrochen werden, dass der Anrufer auflegt, oder dass die Stop-Taste des Anrufbeantworters gedrückt wird. Der Apparat geht dann wieder in den Wartezustand zurück.

Sollen die aufgenommenen Nachrichten abgespielt werden, so wird immer bei der ersten begonnen. Das Abspielen der nächsten Nachricht wird begonnen, wenn die vorhergende fertig abgespielt wurde, oder der Abspielknopf des Anrufbeantworters erneut gedrückt wird. Das Betätigen der Stop-Taste setzt den Anrufbeantworter wieder in den Wartezustand zurück.

Die aufgezeichneten Nachrichten können durch Drücken eines Knopfes gelöscht werden.

[6] Die Aktionen werden jeweils ausgeführt, wenn die Zustände betreten werden; aus Platzgründen wurde hierbei auf die Kennzeichnung `enter` verzichtet.

A.2.2 Elemente des Dialogs

Der Benutzerdialog enthält die folgenden Komponenten:

Ein/Aus Diese Checkbox stellt den Ein- und Ausschalter des Anrufbeantworters dar.

Stop Die Stop-Taste, die den Anrufbeantworter in den Wartezustand zurücksetzt.

Abspielen Mit diesem Knopf werden aufgezeichnete Nachrichten wiedergegeben.

Anzahl In diesem Textfeld zeigt der Anrufbeantworter an, wie viele Nachrichten aufgezeichnet sind. Dieses Feld kann nicht vom Benutzer verändert werden.

PlayTimer Dieser Timer stoppt die Zeit, die für das Abspielen von Ansagen oder Nachrichten benötigt wird. Diese Funktion wird in diesem einfachen Modell benötigt, da für alle diese Vorgänge generell ein Zeitraum von 10 Sekunden angenommen wird.

Löschen Durch Drücken dieses Knopfes werden alle aufgezeichneten Nachrichten gelöscht.

Der Anrufbeantworter reagiert nur auf zwei externe Ereignisse: Das Eingehen eines neuen Anrufs, und das Signal, dass der Anrufer wieder aufgelegt hat. Um diese in der Simulation erzeugen zu können, besitzt der Systemdialog die zwei Knöpfe

Neuer Anruf meldet dem Anrufbeantworter den Beginn eines eingehenden Anrufs

Auflegen beendet einen eingehenden Anruf

Der komplette Dialog der Simulation ist in Abbildung A.10 dargestellt.

Abbildung A.10: Der Simulationsdialog des Anrufbeantworters

A.2.3 Umsetzung in ein Statechart

Die globale Struktur

Der Anrufbeantworter arbeitet einen Vorgang nach dem anderen ab – es ist deshalb naheliegend, ihn als OR-Zustand zu modellieren. Ist die Maschine ausgeschaltet, so reagiert sie auf keine Eingaben; hier bietet es sich an, einen eigenen Zustand einzuführen, der alternativ zum eingeschalteten Zustand liegt. Die gesamte Struktur eines solchen Statecharts ist in Abbildung A.11 dargestellt. Die Beschriftungen der Pfeile sollen hier nur den Überblick vereinfachen; es sind keine eventuell vorhandenen Bedingungen oder Aktionen an ihnen angegeben.

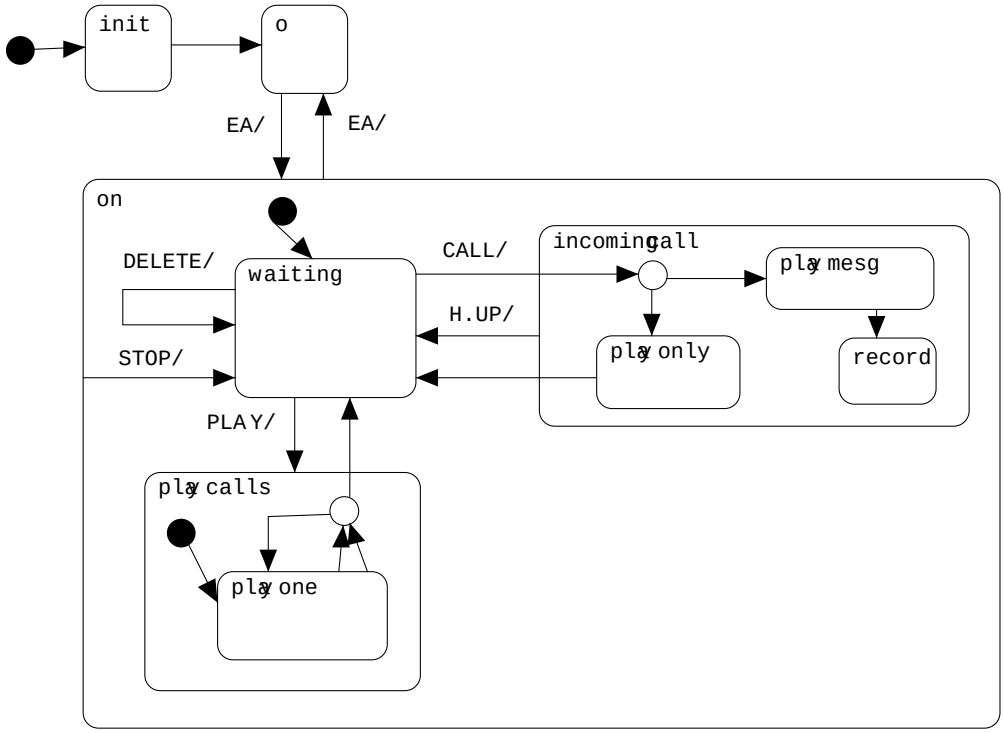


Abbildung A.11: Die Gesamtansicht des Anrufbeantworters

Liste der Variablen und Ereignisse

In diesem Statechart werden die folgenden Variablen benutzt:

MaxAnz Gibt an, wie viele Anrufe der Anrufbeantworter aufzeichnen kann. Prinzipiell besitzt diese Variable die Funktion einer Konstanten; da sie aber an mehreren Stellen im Statechart benutzt wird, verwenden wir eine Variable, um den 'Speicherplatz' des Anrufbeantworters leichter modifizieren zu können. Sie wird lediglich im Zustand `init` gesetzt.

Anrufe In dieser Variablen wird gespeichert, wie viele Nachrichten bereits aufgezeichnet wurden.

PlayNr Diese Variable wird nur im Zustand `play calls` benutzt. Hier gibt sie die Nummer der Nachricht an, die momentan abgespielt wird.

Die Ereignisse in diesem Statechart entsprechen den Dialogkomponenten. Im einzelnen sind dies:

EA das Ereignis, dass der Ein/Ausschalter gedrückt wurde

STOP entspricht der Stop-Taste

PLAY wird vom Knopf „Abspielen“ ausgelöst

played der Timer erzeugt dieses Ereignis, wenn die Zeit abgelaufen ist

DELETE steht für das Drücken des „Löschen“-Knopfes

CALL signalisiert einen neuen eingehenden Anruf – über den Knopf „Neuer Anruf“

HANGUP gibt dem Anrufbeantworter an, dass der Anrufer aufgelegt hat; dies wird durch den Knopf „Auflegen“ ausgeführt

Zustand `init`

Dieser Zustand wird nur am Anfang der Ausführung einmal betreten und sofort wieder verlassen. Er dient dazu, die benutzten Variablen zu initialisieren; dies geschieht beim Betreten durch die folgenden Aktionen:

set!(MaxAnz = 3) Der Speicherplatz des Anrufbeantworters wird auf drei Nachrichten festgelegt.

set!(Anrufe = 0) Es sind noch keine Nachrichten aufgezeichnet

visible!(Anzahl, FALSE) Das Textfeld, das die Anzahl der eingegangenen Anrufe anzeigt, wird deaktiviert, so dass der Benutzer hier keine Werte eingeben kann.

Zustand `off`

Dieser Zustand gibt lediglich an, dass der Anrufbeantworter ausgeschaltet ist. Er wird nach der Inititalisierung betreten. Das einzige Ereignis, das in diesem Zustand zu einer Reaktion des Systems führt, ist das Betätigen des Ein/Ausschalters.

Zustand `on`

In diesem Zustand befindet sich das Statechart, wenn der Anrufbeantworter eingeschaltet ist. Der Default-Einstieg verweist auf den Unterzustand `waiting`, so dass der Automat beim Einschalten immer in diesen Zustand übergeht.

Beim Betreten des Zustands `waiting` werden die folgenden Aktionen ausgeführt:

off!(Timer) Der Timer wird angehalten. Dies hat nur dann eine Bedeutung, wenn durch Drücken von „Stop“ ein Abspielvorgang unterbrochen wurde.

set!(DLG6 = 10) Die Dialogvariable `DLG6`, die zum Timer gehört, wird auf den Wert 10 Sekunden gesetzt. Dies ist in diesem einfachen Modell die Zeit, die sämtliche Ansagen und abgespielten Nachrichten benötigen.

set!(DLG3 = Anrufe) Hierdurch wird die Dialogvariable `DLG3`, die dem Textfeld entspricht, auf den aktuellen Wert der Variablen `Anrufe` gesetzt.

Im Zustand `on` befinden sich einige Pfeile, die den Zustand `waiting` mit anderen Zuständen verbinden. Diese Pfeile sind in der Abbildung A.11 nur durch ihre auslösenden Ereignisse beschriftet. Eine genauere Beschreibung wird in der folgenden Liste gegeben:

CALL Dieser Übergang wird ausgeführt, wenn ein neuer Anruf eingeht, während sich der Anrufbeantworter im Wartezustand befindet. Es ist zu beachten, dass das Ereignis `CALL` ignoriert wird, wenn entweder bereits ein Anruf bearbeitet wird, oder die aufgezeichneten Nachrichten gerade abgespielt werden. Da dieser Pfeil auf einem Universalkonnektor endet, ist der Übergang noch nicht ganz abgeschlossen; für Details der Verzweigung sei jedoch auf Abschnitt A.2.3 verwiesen.

DELETE Wird im Zustand `waiting` der Knopf „Löschen“ gedrückt, so wird dieser Übergang ausgeführt, der den Zustand `waiting` sofort wieder betritt. Dieser Pfeil ist mit der Aktion `set!(Anrufe = 0)` versehen, wodurch die Anzahl der gespeicherten Nachrichten auf `0` gesetzt wird. Beim Betreten von `waiting` wird der Wert des Textfeldes sofort aktualisiert.

PLAY Dieser Übergang wird durch die Bedingung `var?(Anrufe >= 1)` geregelt. Dadurch wird bei dem Ereignis `PLAY` nur dann mit dem Abspielen von Nachrichten begonnen, wenn vorher einige aufgenommen wurden.

STOP Da dieser Pfeil bei dem Oberzustand `on` beginnt, wird er beim Drücken des Knopfes „Stop“ ausgeführt, unabhängig davon, in welchem Unterzustand sich das System gerade befindet.

H.UP Dieser Übergang beendet einen eingehenden Anruf. Er wird entweder ausgeführt, wenn der Anrufer auflegt, bevor die Aufnahme begonnen wurde, oder wenn der Anrufer seine Nachricht vollständig aufgesprochen hat.

Zustand *incoming calls*

Dieser Zustand entspricht der Bearbeitung eines eingehenden Anrufs. Eine detaillierte Darstellung findet sich in Abbildung A.12.

Betreten wird dieser Zustand mit dem Übergang `CALL`; je nach dem, ob der Speicher noch Platz hat (`Anruf` $<$ `MaxAnz`) oder schon voll ist (`Anruf` $==$ `MaxAnz`), wird nun der Zustand `play mesg` oder `play only` betreten. In beiden Fällen wird der Timer gestartet – eigentlich wird auch eine Ansage abgespielt, aber dies wurde nicht in das Modell miteinbezogen.

Ist der Speicher bereits voll, beendet das System nach dem Abspielen den Anruf und geht wieder in den Ruhezustand über. Ansonsten wird nach der Ansage, wenn

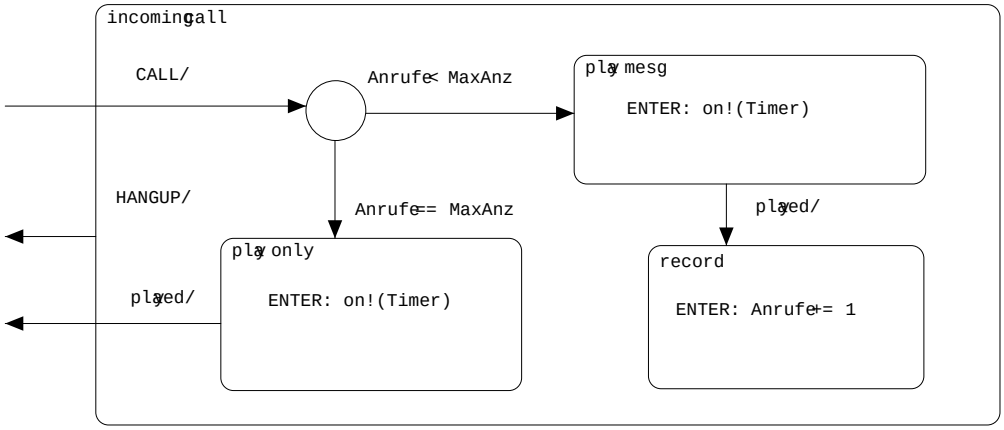


Abbildung A.12: Verhalten bei einem ankommenden Anruf

der Anrufer noch nicht aufgelegt hat, die Aufnahme begonnen. Beim Betreten des Zustands `record` wird die Anzahl der aufgenommenen Nachrichten um eins erhöht. Die Aufnahme wird beendet, wenn der Zustand `incoming call` durch das Ereignis `HANGUP` wieder verlassen wird.

Zustand `play calls`

Wenn im Wartezustand der Knopf „Abspielen“ gedrückt wird, und bereits Nachrichten aufgenommen wurden, so wird dieser Zustand betreten. Der Zustand ist in Abbildung A.13 dargestellt.

Beim Betreten des Zustands `play calls` wird die Variable `PlayNr`, die angibt, welche Nachricht abgespielt wird, mit dem Wert 1 initialisiert. Der Default-Einstieg verweist auf den Zustand `play one`, und beim Betreten dieses Zustands wird der Timer auf den Wert 10 gesetzt und gestartet.

Wenn die Taste „Abspielen“ gedrückt wird (Ereignis `PLAY`) oder der Timer abgelaufen ist (Ereignis `played`), wird der Zustand `play one` verlassen und dabei die Zählvariable um eins erhöht. Abhängig davon, ob noch weitere Nachrichten abzuspielen sind (`PlayNr` $<$ `Anrufe`) oder gerade die letzte abgespielt wurde (`PlayNr` $>=$ `Anrufe`), wird der Zustand `play one` gleich wieder betreten, oder der Zustand `play calls` verlassen. Im zweiten Fall geht das Statechart wieder in den Wartezustand über.

A.2.4 Möglichkeiten der Verfeinerung

Wie bereits mehrfach erwähnt, ist dieses Modell sehr einfach gehalten. Dementsprechend gibt es mehrere Punkte, in denen das Modell weitergeführt werden könnte. Einerseits gehört dazu das Eingeben der Ansagetexte, andererseits eine feinere Behandlung der aufgenommenen Nachrichten. So bieten die meisten Anrufbeantworter die Möglichkeit, neue von alten Nachrichten zu unterscheiden und einzelne Nachrichten zu löschen. Um dies jedoch komfortabel zu modellieren, wären Felder von Variablen notwendig, die in *SCed* nicht implementiert wurden. Andere Erweiterungen bestünden

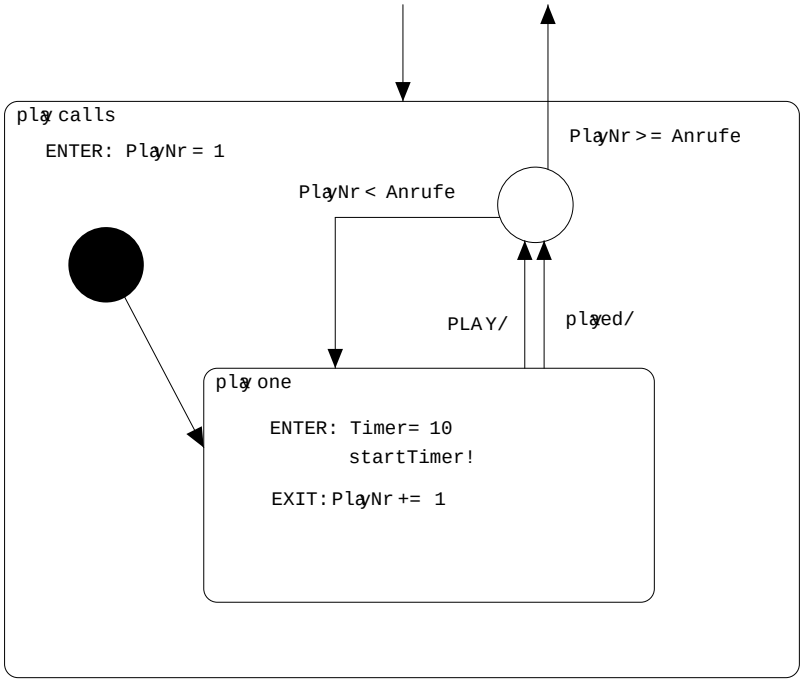


Abbildung A.13: Das Abspielen von Anrufen

in Spezialfunktionen, die einzelne Anrufbeantworter bieten, wie etwa dem Aufnehmen von Notizen des Benutzers oder einer Fernabfrage.

Das wahre Glück, Du Menschenkind,
oh, glaube doch mitnichten,
daß es erfüllte Wünsche sind –
es sind erfüllte Pflichten!
(Deutsches Sprichwort)

Anhang B

Pflichtenheft

Dieses Dokument wurde vor Entwurfsbeginn entworfen um die Zielkriterien, welche unser Programm implementieren sollte festzulegen. Insbesondere bei den Kann-Kriterien tauchten während der Arbeit Gründe auf, einige davon nicht zu implementieren.[1]

B.1 Zielbestimmung

B.1.1 Muss-Kriterien

Es soll ein Programm zur Erstellung von Statecharts entworfen werden. Im einzelnen sollen die folgenden Programmteile dazu implementiert werden:

- Grafischer Editor mit mehrere Ansichten zum Editieren der Statecharts
- Rudimentärer Dialog-Editor
- Simulator, der die Funktion der Dialoge mit verbundenen Ereignissen abarbeitet – Quelltext wird in gewissem Umfang beachtet.
- Verifikator zum Finden einfacher Fehler
- Quelltextgenerierung des erstellten Dialogmodells in Java
- Mehrfach Instantiierung von Teilen des Statecharts, ohne dass diese kopiert werden müssen.

  (Diese Anforderung wurde in Absprache fallengelassen.)
- Statecharts drucken

[1] vgl. Abschnitt 6.9

B.1.2 Kann-Kriterien

Die folgenden Funktionen können zusätzlich in das Programm eingebaut werden, falls die Zeit dies erlaubt:

- Objektbezogene Undo/Redo-Funktion
- Komplexere Aktionen (Makros) zusammenfassen
- statische Bedingung-Aktion-Gruppen in Zuständen
- Bedingungen bei Default-Einstieg zulassen
- Verifikation von komplizierteren Verhältnissen, wie z. B. Races
- Diagramm-Konnektoren
- verzögertes Auslösen von Ereignissen
- beide Formen von Zeitmodell (synchron/asynchron) – siehe dazu die Diskussion dieses Themas in der Literatur in Abschnitt 2.8.
- Ereignisse (wie bereits Bedingungen und Aktionen) aus Java-Quelltext generieren (etwa zum Abfangen von Exceptions)
- Endzustände, die das System beenden und nicht nur reaktionslos machen

B.1.3 Abgrenzungskriterien

Das Programm soll keinen vollständigen grafischen Dialog-Editor bieten. Außerdem werden weder Activity-Charts noch die in Statemate implementierte physikalische Sicht angeboten. In der Simulation werden die durch Java-Quelltext eingegebenen Bedingungen und Aktionen nicht ausgeführt.

B.2 Einsatz

B.2.1 Anwendungsbereiche

Hilfen und logische Konstruktion von interaktiven Systemen; Testen des Modells und Erzeugen von Quelltext-Gerüsten.

B.2.2 Zielgruppe

Dialog- und Benutzerschnittstellenentwickler

B.3 Umgebung

B.3.1 Software

Zum Betrieb ist ein lauffähiges Sun JRE mit einer Version $\geq$ 1.2.2 nötig. Weiterhin benötigt man eine grafische Oberfläche inklusive Windowmanager. Damit ist das Programm auf Sun Workstations unter Solaris sowie x86-PCs unter Linux ausführbar. Ein Betrieb unter Windows ist wohl möglich, wird aber aufgrund der Unterschiede im OS nicht garantiert. Weiterhin müssen für die Übersetzung exportierte Statecharts und die Installationsprogramme das Programm `make` und die `bash`-Shell.

B.3.2 Hardware

Die Rechner müssen mit einem grafischen Bildschirm, Tastatur und Maus ausgestattet sein.

Sapienti sat.[1]
(Plautus, „Der Perser“ 729)

Anhang C

Grammatik des Datenformats

In diesem Anhang wird die komplette Grammatik der Sprache beschrieben, die zum Speichern der Statecharts verwendet wird.[2]

command	::=	createState | createKonnektor | insertState | addView | addPartitiner | createArrow | createEvent | createAction | addCondition | addAction | addEvent | createText | addText
createState	::=	“CREATE” stateType identification
createKonnektor	::=	“CREATE” konnType identification
createDialog	::=	“CREATE” dlgType id name [“VISIBLE”] [“SYSTEM”]
insertState	::=	“INSERT” id “IN” stateId offset
addView	::=	“ADD” “VIEW” stateId depth zoom [“GRID”] [“HOOKS”] [“LABEL”] [“ID”]
addPartitioner	::=	“ADD” “PARTITIONER” stateId first second
createArrow	::=	“CREATE” “ARROW” (create1-1 | create1-N | createN-1)
createEvent	::=	“CREATE” “EVENT” eventId text
createAction	::=	“CREATE” “ACTION” actionId (clearHistAct | deepClearHistAct | setDialogVisible | incVarAct| setDialogOnAct | setDialogOffAct | throwAct | javaAct)
createText	::=	“CREATE” “TEXT” id offset text
addCondition	::=	“ADD” “CONDITION” “TO” arrowId condition
addAction	::=	“ADD” “ACTION” actionId “TO” target
addEvent	::=	“ADD” “EVENT” eventId “TO” arrowId
addText	::=	“ADD” “TEXT” id “TO” stateId
identification	::=	id name size
stateType	::=	“STATE” | “ORSTATE” | “ANDSTATE” | “ENDSTATE”
konnType	::=	“DEFAULT” | “UNIVERSAL” | “HISTORY | “DEEPHISTORY”
dlgType	::=	“BUTTON” | “CHECKBOX” | “TIMER” | “TEXTFIELD” | “RADIOBUTTON” | “POPUP”
create1-1	::=	“1-1” arrowId onlySection
create1-N	::=	“1-N” arrowId oneStart connectorHook endSections
createN-1	::=	“N-1” arrowId startSections connectorHook oneEnd

[1] Für den Verständigen genug.
[2] vgl. Abschnitt 6.8

```
        condition   ::=  basicCondition | andCondition | orCondition | notCondition

      clearHistAct  ::=  “CLEARHIST” stateId
      clearHistAct  ::=  “clearHist!” “(” stateId “)”
  deepClearHistAct  ::=  “DEEPCLEAR” stateId
  deepClearHistAct  ::=  “clearDeep!” “(” stateId “)”
setDialogVisibleAct ::=  “VISIBLE” dlgCompId visible
setDialogVisibleAct ::=  “visible!” “(” dlgCompId “,” visible “)”
   setDialogOnAct   ::=  “DLGON” dlgCompId
   setDialogOnAct   ::=  “on!” “(” dlgCompId “)”
   setDialogOffAct  ::=  “DLGOFF” dlgCompId
   setDialogOffAct  ::=  “off!” “(” dlgCompId “)”
       setVarAct    ::=  “SETVAR” varname varvalue
       setVarAct    ::=  “set!” “(” varname “=” varvalue “)”
       incVarAct    ::=  “INCVAR” varname varvalue
       incVarAct    ::=  “inc!” “(” varname “=” varvalue “)”
       throwAct     ::=  “THROW” eventId
       throwAct     ::=  “throw!” “(” eventId “)”
        javaAct     ::=  “JAVA” popupId filename
        javaAct     ::=  “java!” “(” popupId “,” filename “)”

     andCondition   ::=  “{” condition { “AND” condition } “}”
      orCondition   ::=  “{” condition { “OR” condition } “}”
     notCondition   ::=  “{” “NOT” condition “}”
   basicCondition   ::=  inStateCond | hasHistoryCond | javaCond | varCond
     inStateCond    ::=  “IN” “STATE” stateId
     inStateCond    ::=  “in?” “(” stateId “)”
  hasHistoryCond    ::=  “HAS” “HISTORY” stateId
  hasHistoryCond    ::=  “hasHist?” “(” stateId “)”
      javaCond      ::=  “JAVA” javaCommand
      javaCond      ::=  “java?” “(” javaCommand “)”
       varCond      ::=  “VARIABLE” varname operator varvalue
       varCond      ::=  “var?” “(” varname operator varvalue “)”

      oneStart      ::=  section
       oneEnd       ::=  section
   startSections    ::=  “{” section { section } “}”
   endSections      ::=  “{” section { section } “}”
      varvalue      ::=  double | integer | string | varname

     onlySection    ::=  “{” hook hook { hook } “}”
       section      ::=  “{” hook { hook } “}”
  connectorHook     ::=  hook

        target      ::=  targetState | targetArrow | targetDialog
   targetState      ::=  “STATE” stateId “ON” ( “ACTIVATION” | “DEACTIVATION”)
   targetArrow      ::=  “ARROW” arrowId
   targetDialog     ::=  “DIALOG” dlgCompId
        hook        ::=  “<” anchorStateId offset [textId] “>”
```

varname	::=	identifier
zoom	::=	double
size	::=	dimension
offset	::=	point
first	::=	point
second	::=	point
depth	::=	integer
name	::=	string
text	::=	string
javaCommand	::=	string
filename	::=	string
id	::=	integer
eventId	::=	integer
actionId	::=	integer
textId	::=	integer
timerId	::=	integer
dlgCompId	::=	integer
popupId	::=	integer
anchorStateId	::=	integer
stateId	::=	integer
arrowId	::=	integer
visible	::=	boolean

dimension	::=	“[” double “,” double “]”
point	::=	“(” double “,” double “)”
double	::=	integer “.” integer
string	::=	““” { character } “””
identifier	::=	character { character }
integer	::=	digit { digit }
digit	::=	“0” | . . . | “9”
boolean	::=	“TRUE” | “FALSE”
operator	::=	“==” | “>=” | “<=” | “<” | “>” | “<>”

The Dean glared.
‘That’s not magic!’ he snapped.
‘That’s just … engineering!’
(Terry Pratchett, „Interesting Times“)

Anhang D

Programmdokumentation

D.1 Package score

D.1.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.start*

D.1.2 Klasse start

extends *java.lang.Object*
score - Hauptklasse. Ruft die Steuerklasse auf und startet dadurch das Programm.

Methoden

+$ main (*String[]*) → *void*
Aufruf des Programms als Application.

D.2 Package score.AbstractModell

D.2.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.util.Observable*
    ↪ *score.AbstractModell.AbstractModell*
    ↪ *score.AbstractModell.MarkableTyp*

D.2.2 Klasse AbstractModell

(abstrakte Klasse)
extends *score.util.Observable*
implements *score.util.Observer*

Das abstrakte Obermodell des grafischen und logischen Modells

Attribute

–$ marked:*Vector* – Vektor, der sich alle markierten Objekte merkt.

+$ markingOperation:*boolean* – Flag zeigt an, ob die letzte Operation die Markierung geändert hat

+$ doRefresh:*boolean* – Gibt an, ob ein Refresh durchgeleitet wird. Mit dieser Variablen kann man verhindern, dass beim Aufruf von refresh() tatsächlich ein Refresh durchgeführt wird, falls mehrere Änderungen gleichzeitig durchgeführt werden sollen.

Konstruktoren

\+ AbstractModell ()
Konstruktor Löscht den Vektor mit den markierten Objekten!

Methoden

+$ getMarkedObject () → *MarkableTyp*
Rückgabewert: das erste markierte Objekt oder null, wenn kein Objekt markiert ist.
Liefert das markierte Objekt

+$ getMarkedObjects () → *Enumeration*
Rückgabewert: Aufzählung aller markierten Objekte oder null, wenn kein Objekt markiert ist
Liefert eine Enumeration aller markierten Objekte

+$ nothingMarked () → *boolean*
Rückgabewert: true, wenn Objekte markiert sind, sonst false
Zeigt an, ob ein Objekt markiert ist

+$ isMarked (*MarkableTyp*) → *boolean*
Rückgabewert: true, wenn m markiert ist, sonst false
Prüft ob das übergebene Objekt markiert ist
Parameter:

- m: Zu prüfendes Objekt

+$ countMarkedObjects () → *int*
Gibt die Anzahl der markierten Objekte zurück.

\+ setMarkedObject (*MarkableTyp*) → *void*
Setzt das übergebene Objekt als einziges markiertes
Parameter:

- m: Objekt, das markiert ist.

\+ setMarkedObjects (*MarkableTyp[]*) → *void*
Setzt den Array als markierte Objekte.
Parameter:

- m: Array von Objekten, die markiert sein sollen.

\+ addMarkedObject (*MarkableTyp*) → *void*
Fügt das übergebene Objekt zu den markierten hinzu
Parameter:

- m: Objekt das auch markiert sein soll

\+ addMarkedObjects (*MarkableTyp[]*) → *void*
Fügt ein Array von Objekten den markierten Objekten hinzu
Parameter:

- m: Array von Objekten die auch markiert sein sollen

\+ unmarkObject (*MarkableTyp*) → *void*

Setzt das Objekt m als nicht markiert

\+ clearMarkedObjects () → *void*
Demarkiert alle Objekte

\+ refresh (*Observable*) → *void*
Benachrichtigt alle Observer, dass sich das Modell geändert hat.

\+ refresh () → *void*
Benachrichtigt alle Observer, dass sich das Modell geändert hat.

\+ update (*Observable*, *Object*) → *void*
Implementation des Observer-Interface

\+ getAll () → *Vector*
Liefert einen Vector aller Elemente

\+ dumpMemory () → *String*
Rückgabewert: String des Memory-Dumps
Gibt einen MemoryDump zurück

\+ toString () → *String*
überschreibt die Methode der Klasse *Object*.
Rückgabewert: String des Memory-Dumps
Gibt einen MemoryDump zurück

\# dumpOwnMemory () → *String*
Rückgabewert: String des Memory-Dumps
Gibt einen MemoryDump zurück

D.2.3 Klasse MarkableTyp

(abstrakte Klasse)
extends *score.util.Observable*
Abstrakte Oberklasse aller Objekte, die markiert werden können

Attribute

\# type:*int* – Der Typ dieses MarkableTyp

\+$ NOTHING:*int* CONST – Konstante für ungültigen Typ

\+$ STATE:*int* CONST – Konstante für einen Zustand (atomar, OR, AND)

\+$ CONNECTOR:*int* CONST – Konstante für einen Konnektor (universal, history)

\+$ ARROW:*int* CONST – Konstante für eine Verbindung

\+$ PARTITIONER:*int* CONST – Konstante für einen Partitionsgrenzentrenner

\+$ HOOK:*int* CONST – Konstante für einen grafischen Knotenpunkt

\+$ GRAF_TEXT:*int* CONST – Konstante für einen Text

\+$ BUTTON:*int* CONST – Konstante für die Dialogkomponente Button

\+$ TEXT:*int* CONST – Konstante für die Dialogkomponente Textfeld

\+$ TIMER:*int* CONST – Konstante für die Dialogkomponente Timer

\+$ CHECKBOX:*int* CONST – Konstante für die Dialogkomponente Checkbox

\+$ RADIOBUTTON:*int* CONST – Konstante für die Dialogkomponente RadioButton

\+$ POPUP:*int* CONST – Konstante für die Dialogkomponente Popup

\+$ VARIABLE:*int* CONST – Konstante für eine Variable

\+$ VARIABLE_INTEGER:*int* CONST – Konstante für eine Variable vom Typ Integer

\+$ VARIABLE_DOUBLE:*int* CONST – Konstante für eine Variable vom Typ Double

\+$ VARIABLE_STRING:*int* CONST – Konstante für eine Variable vom Typ String

\+$ doRefresh:*boolean* – Gibt an, ob ein refresh durchgeleitet werden soll

Methoden

\+ getType () → *int*

Rückgabewert: int des Typs dieser Instanz

Gibt den Typ dieser Instanz zurück

+ isGrafTyp () → *boolean*

Rückgabewert: true, wenn es ein grafisches Objekt ist, sonst false

Gibt zurück, ob dieser Typ ein grafisches Objekt ist

+ isDialogTyp () → *boolean*

Rückgabewert: true, wenn es ein Dialogobjekt ist, sonst false

Gibt zurück, ob dieser Typ ein Dialogobjekt ist

+ toString () → *String*

überschreibt die Methode der Klasse *Object*.
Gibt eine Stringrepräsentation zurück

+ refresh (*Observable*) → *void*

Führt einen refresh durch

+ refresh () → *void*

Führt einen refresh durch

D.3 Package score.Aktionen

D.3.1 Klassenhierarchie

· *java.lang.Object*
- ↪ *score.Aktionen.AktionEdit*
- ↪ *score.Aktionen.CalledAction*
- ↪ *score.LogSCM.LogComponent*
  - ↪ *score.Aktionen.AktKomplex*
    - ↪ *score.Aktionen.ExportAktion*
  - ↪ *score.Aktionen.Aktion*
    - ↪ *score.Aktionen.ClearHistoryAktion*
    - ↪ *score.Aktionen.DeepClearHistoryAktion*
    - ↪ *score.Aktionen.JavaAktion*
    - ↪ *score.Aktionen.SetDialogOnAktion*
    - ↪ *score.Aktionen.SetDialogVisibleAktion*
    - ↪ *score.Aktionen.SetVarAktion*
      - ↪ *score.Aktionen.IncVarAktion*
    - ↪ *score.Aktionen.ThrowAktion*
- ↪ *score.util.Observable*
  - ↪ *score.Aktionen.AktionDialog*
  - ↪ *score.Aktionen.AktionPropertyChange*
    - ↪ *score.Aktionen.AktionProperty*

D.3.2 Interface Actor

Dieses Interface wird von den Klassen implementiert, die Aktionen ausführen können. Es ermöglicht den einheitlichen Zugriff auf diese Aktionen, sowie die Anfrage, um welcen Typ es sich handelt.

Attribute

+$ ACTOR_STATE:*int* CONST – Konstante, falls dieser Actor ein Zustand ist.

+$ ACTOR_ARROW:*int* CONST – Konstante, falls dieser Actor ein Pfeil ist.

+$ ACTOR_DIALOG:*int* CONST – Konstante, falls dieser Actor ein Dialogelement ist.

+$ ACTIVATE:*int* CONST – Konstante, über die bei Zuständen auf die Aktionen beim Aktivieren zugegriffen werden kann.

+$ DEACTIVATE:*int* CONST – Konstante, über die bei Zuständen auf die Aktionen beim Deaktivieren zugegriffen werden kann.

Methoden

+ getActorType () → *int*
Rückgabewert: int entsprechend der Klassenkonstante
Gibt den Typ dieses Actors zurück.

+ getCalledAction (*int*) → *CalledAction*
Gibt eine komplexe Aktion zurück, die für die angegebene Position (Klassenkonstante ACTIVATE oder DEACTIVATE) alle Aktionen enthält.

+ getCalledAction () → *CalledAction*
Liefert eine komplexe Aktion aller Aktionen.

+ doesAction (*Aktion*) → *boolean*
Rückgabewert: true, falls diese Aktion enthalten ist.
Gibt zurück, ob dieser Actor die übergebene Aktion ausführt.

+ removeAction (*Aktion*) → *void*
Entfernt die übergebene Aktion aus diesem Actor.

+ setChanged () → *void*
Teilt dem Actor mit, dass sich seine Aktionen verändert haben.

+ setObserversChanged () → *void*
Teilt dem Actor mit, dass sich seine Aktionen verändert haben. Der Actor wird auch für alle Observer den Status auf *changed* setzen.

+ notifyObservers () → *void*
Ruft die Methode notifyObservers der Klasse Observable auf.

D.3.3 Interface DialogAktion

Regelt den Zugriff auf Dialogkomponenten von Aktionen die sie auf solche beziehen

Methoden

+ setDlgKomponente (*DlgKomponente*) → *void*
Setzt die Dialogkomponente für die Aktion

+ getDlgKomponente () → *DlgKomponente*
Liefert die Dialogkomponente dieser Aktion

D.3.4 Interface EventHolder

Dieses Interface dient dazu, Ereignisse zu erzeugen (d. h. in der implementierenden Klasse abzulegen) und die erzeugten Ereignisse zu holen.

Methoden

+ getEvents () → *Vector*
Gibt einen Vector der erzeugten Ereignisse zurück.

+ storeEvent (*Object*) → *void*
Speichert das übergebene Ereignis. Es wird hier nur eine Instanz der Klasse *Object* verlangt, weil es je nach Zusammenhang (Simulation oder exportierter Quelltext) um die Klasse *Ereignis* oder *String* geht.

D.3.5 Interface HistoryAktion

Dieses Interface bietet den Zugriff auf den logischen OR-Zustand der implementierenden Aktion.

Methoden

+ getState () → *ZustOR*
  Gibt den dieser Aktion zugeordnenten OR-Zustand zurück.
+ setState (*ZustOR*) → *void*
  Setzt den OR-Zustand für diese Aktion.

D.3.6 Klasse Aktion

(abstrakte Klasse)
extends *score.LogSCM.LogComponent*
implements *java.lang.Comparable*

Diese abstrakte Klasse stellt eine atomare Aktion dar.

Attribute

+$ ACT_JAVA:*int* CONST – Konstante, falls es sich um eine Java-Aktion handelt.
+$ ACT_CLEAR_HISTORY:*int* CONST – Konstante, falls es sich um eine ClearHistory-Aktion handelt.
+$ ACT_CLEAR_DEEP:*int* CONST – Konstante, falls es sich um eine ClearDeepHistory-Aktion handelt.
+$ ACT_THROW:*int* CONST – Konstante, falls es sich um eine Throw-Aktion handelt.
+$ ACT_SET_VAR:*int* CONST – Konstante, falls es sich um eine SetVariable-Aktion handelt.
+$ ACT_INC_VAR:*int* CONST – Konstante, falls es sich um eine InVar-Aktion handelt.
+$ ACT_SET_DIA_VISIBLE:*int* CONST – Konstante, falls es sich um eine SetDialogVisible-Aktion handelt.
+$ ACT_DIALOG_ON:*int* CONST – Konstante, falls es sich um eine StartTimer-Aktion handelt.
+$ TYPES:*String* CONST – Konstante mit den Namen der Aktionen

Konstruktoren

Aktion (*int*)
  Erzeugt eine neue atomare Aktion vom angegebenen Typ.

Methoden

+$ setSimModell (*SimSCModell*) → *void*
  Setzt das SimulationsModell, auf das sich die Aktionen beziehen.
+ getType () → *int*
  Gibt den Typ dieser Aktion als Integer zurück.
+ getID () → *int*
  Gibt die ID dieser atomaren Aktion zurück.
+$ getAll () → *Vector*
  Gibt einen Vector aller atomaren Aktionen zurück.
+$ getAction (*int*) → *Aktion*
  Gibt die atomare Aktion mit dem angegebenen Identifier zurück.
+$ getActors (*int*) → *Vector*
  Gibt einen Vector der Actors für die atomare Aktion mit dem angegebenen Identifier zurück.
+ getActors () → *Vector*

Gibt einen Vector der Actors dieser atomaren Aktion zurück.

+ hasActors () → *boolean*
  Gibt zurück, ob diese atomare Aktion Actors eingetragen hat.
+ addActor (*Actor*) → *void*
  Fügt zu dieser atomaren Aktion den übergebenen Actor hinzu.
+ removeActor (*Actor*) → *void*
  Entfernt den übergebenen Actor aus der Liste der Actors dieser atomaren Aktion.
+ compareTo (*Object*) → *int*
  Implementiert das Comparable-Interface.
+ delete () → *void*
  überschreibt die Methode der Klasse *Observable*.
  Löscht diese Aktion, d. h. sie wird aus dem Hashtable entfernt.
+ equals (*Object*) → *boolean*
  überschreibt die Methode der Klasse *Object*.
  Vergleicht den Identifier
+ execute () → *boolean*
  Führt diese Aktion aus. Muss entsprechend implementiert werden.
+ toString () → *String*
  überschreibt die Methode der Klasse *Object*.
  Stellt diese Aktion als String dar; wird für die Ausgabe in den Listen benötigt und muss entsprechend implementiert werden.
+ save (*EasyFileWriter*) → *void*
  Speichert diese atomare Aktion in den übergebenen EasyFileWriter.
+$ exportAll (*EasyFileWriter*) → *void*
  Exportiert alle Aktionen in den EasyFileWriter. Dabei wird eine `switch`-Anweisung geschrieben, die mit dem Inhalt der atomaren Aktionen mittels exportQT gefüllt wird.
+ exportQT (*EasyFileWriter*) → *void*
  Diese abstrakte Methode muss von den Aktionen entsprechend implementiert werden, um den entsprechenden Quelltext in den EasyFileWriter zu exportieren.

D.3.7 Klasse AktionDialog

extends *score.util.Observable*
implements *java.awt.event.ActionListener,*
score.util.Observer,
javax.swing.event.ListSelectionListener,
java.awt.event.WindowListener

Diese Klasse stelle einen Dialog zum Auswählen von Aktionen zur Verfügung

Attribute

+$ undo:*UndoManager* – Globaler UndoManager

Konstruktoren

+ AktionDialog (*LogDModell, WindowListener, String*)
  Konstruktor erzeugt einen neuen Dialog mit logischem Modell ldm
+ AktionDialog (*LogDModell*)
  Erschafft einen AktionDialog zum übergebenen LogDModell als globale Aktionen-Liste.

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*

Die Observables wurden ersetzt

+ actionPerformed (*ActionEvent*) → *void*
Für den ActionListener

+ dispose () → *void*
Löscht diesen AktionDialog.

+ update (*Observable, Object*) → *void*
Für den Observable

+ valueChanged (*ListSelectionEvent*) → *void*
Die Listenselektion hat sich geändert

D.3.8 Klasse AktionEdit

extends *java.lang.Object*
implements *java.awt.event.ActionListener,*
score.util.Observer,
javax.swing.event.ListSelectionListener,
java.awt.event.WindowListener

Diese Klasse stellt eine Lister alle Aktionen eines Objekt bzw. einer komplexen Aktion

Attribute

+$ undo:*UndoManager* – Globaler UndoManager

Konstruktoren

+ AktionEdit (*LogDModell, AktKomplex, String*)
Konstruktor

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*
Ersetzt das Observable

+ actionPerformed (*ActionEvent*) → *void*
Wertet ButtonEvents aus

+ update (*Observable, Object*) → *void*
Observerfunktion

+ valueChanged (*ListSelectionEvent*) → *void*
Listenselektion hat sich geändert

D.3.9 Klasse AktionProperty

extends *score.Aktionen.AktionPropertyChange*
Ändert die Eigenschaften einer Aktion

Konstruktoren

+ AktionProperty (*LogDModell*)
Konstruktor

Methoden

init () → *void*
überschreibt die Methode der Klasse *AktionPropertyChange*.
Initialisiert diese Klasse und setzt den ActionListener für die ComboBox

+ update (*Observable, Object*) → *void*

überschreibt die Methode der Klasse *AktionPropertyChange*.
Observer: sendet die neue Aktion weiter

\# initializeComponents (*int*) → *void*

überschreibt die Methode der Klasse *AktionPropertyChange*.
Dummy

\+ actionPerformed (*ActionEvent*) → *void*

überschreibt die Methode der Klasse *AktionPropertyChange*.
Reagiert auf die ButtonEvents

D.3.10 Klasse AktionPropertyChange

extends *score.util.Observable*
implements *java.awt.event.ActionListener,*
score.util.Observer

Erstellt den Aktionseigenschaftsdialog

Attribute

\# ldm:*LogDModell* – Referenz auf das logische Dialogmodell

\# frame:*JFrame* – Das zugehörige Frame

\# typeSelect:*JComboBox* – Die Aktionstyp-Combobox

\# javaEditor:*JEditorPane* – Ein Editorfeld für den Java-SourceCode

\# varName:*JTextField* – Textfeld für Variablenname und Variablenwert

\# varValue:*JTextField* – Textfeld für Variablenname und Variablenwert

\# checkbox:*JCheckBox* – generische Checkbox

\# historyCon:*JList* – Liste aller Zustände mit Historyeinstiegen

Konstruktoren

\# AktionPropertyChange (*LogDModell*)
Konstruktor, der eine neue Aktion behandelt (NEW)

\+ AktionPropertyChange (*LogDModell, Aktion*)
Konstruktor, der eine vorhandene Aktion editiert

Methoden

\# init () → *void*
Initialisiert die Klasse

\+ update (*Observable, Object*) → *void*
Observerfunktion setzt die Aktionen

\# sendAction (*Aktion*) → *void*
Sendet die generierte Aktion an die Observer

\# initializeComponents (*int*) → *void*
Initialisiert die Componenten für eine Aktion vom Typ type

\# updateFrame () → *void*
Zeichnet den Frame neu

\+ actionPerformed (*ActionEvent*) → *void*
Wertet ButtonEvents aus

D.3.11 Klasse AktKomplex

extends *score.LogSCM.LogComponent*
Diese Klasse stellt eine komplexe Aktion dar. Sie enthält mehrere atomare Aktionen, die mit AND verbunden sind, d. h. sie werden alle nacheinander ausgeführt.

Konstruktoren

\+ AktKomplex (*Actor*)

erzeugt eine neue komplexe Aktion zum übergebenen Actor. Sie enthält anfangs noch keine atomaren Aktionen.

Methoden

\+ delete () → *void*

überschreibt die Methode der Klasse *Observable*.
Löscht diese komplexe Aktion - die in ihr enthaltenen atomaren Aktionen bleiben davon unberührt.

\+ getActor () → *Actor*

Gibt den Actor dieser komplexen Aktion zurück.

\+ getIdent () → *String*

Gibt einen String zurück, der diese komplexe Aktion für den Quelltextexport eindeutig identifiziert.

\+ getID () → *int*

Gibt den Identifier dieser komplexen Aktion zurück.

+$ getAktKomplex (*int*) → *AktKomplex*

Gibt die komplexe Aktion mit dem angegebenen Identifier zurück.

\+ size () → *int*

Gibt die Anzahl der atomaren Aktionen in dieser komplexen Aktion an.

\+ getActions () → *Vector*

Gibt einen Vector der enthaltenen atomaren Aktionen zurück.

\+ elements () → *Enumeration*

Gibt eine Enumeration der enthaltenen atomaren Aktionen zurück.

\+ contains (*Aktion*) → *boolean*

Gibt an, ob die übergebene Aktion in dieser komplexen Aktion enthalten ist.

\+ execute (*Actor*) → *boolean*

Führt diese komplexe Aktion aus. Dabei werden nacheinander alle enthaltenen Aktionen durch ihre Methode execute ausgeführt.

\+ addAction (*Aktion*) → *void*

Fügt eine atomare Aktion zu dieser komplexen Aktion hinzu. Dabei kann es vorkommen, dass eine atomare Aktion mehrfach auftritt.

\+ addAll (*AktKomplex*) → *void*

Fügt zu dieser komplexen Aktion alle atomaren Aktionen der übergebenen komplexen Aktion hinzu.

\+ removeAction (*Aktion*) → *void*

Entfernt eine atomare Aktion aus dieser komplexen Aktion. Auch, wenn sie mehrfach enthalten war, ist sie danach nicht mehr enthalten.

\+ updateTranslators (*Hashtable[]*) → *void*

Fügt zu den übergebenen übersetzungstabellen die Ereignisse und Variablen hinzu, die in den atomaren Aktionen dieser komplexen Aktion benutzt werden.

\+ translateInto (*Hashtable[], AktKomplex*) → *void*

Fügt eine Übersetzung dieser komplexen Aktion in die übergebene komplexe Aktion ein. ThrowAktionen, SetVar und IncVarAktionen werden neu erschaffen, wenn das Ereignis bzw. eine der Variablen ersetzt werden müssen, ansonsten werden die alten atomaren Aktionen übernommen.

\+ equals (*AktKomplex*) → *boolean*

Gibt zurück, ob die enthaltenen atomaren Aktionen dieser und der übergebenen komplexen Aktion übereinstimmen. (Dies ist essentiell die Methode Vector.equals().)

\+ save (*EasyFileWriter, String*) → *void*

Speichert diese komplexe Aktion in den EasyFileWriter.

Parameter:

- target: String, der angibt, bei welchem Actor und an welche Stelle diese komplexe Aktion beim Laden wieder eingefügt werden muss.

+$ exportAll (*EasyFileWriter*) → *void*

Exportiert alle komplexen Aktionen in den EasyFileWriter. Dafür wird die Methode executeCompAction(String) in das File geschrieben.

\+ exportQT (*EasyFileWriter*) → *void*

Exportiert eine komplexe Aktion in Form einer `case`-Anweisung.

D.3.12 Klasse CalledAction

extends *java.lang.Object*
Diese Klasse stellt eine aufgerufene Aktion dar. Sie enthält neben der eigentlichen Aktion auch den aufrufenden Actor.

Konstruktoren

\+ CalledAction (*AktKomplex, Actor*)

Erzeugt eine neue CalledAction aus der übergebenen Aktion mit dem entsprechenden Actor.

Methoden

\+ execute () → *void*

Führt die Aktion aus.

\+ getActions () → *AktKomplex*

Gibt die komplexe Aktion dieser CalledAction zurück.

\+ isEmpty () → *boolean*

Gibt an, ob diese CalledAction leer ist.

D.3.13 Klasse ClearHistoryAktion

extends *score.Aktionen.Aktion*
implements *score.Aktionen.HistoryAktion*

Diese Klasse spezialisiert die abstrakte Klasse Aktion als ClearHistoryAktion.

Konstruktoren

\+ ClearHistoryAktion (*ZustOR*)

Erschafft eine neue Aktion, die die History des übergebenen Zustands löscht.

Methoden

\+ getState () → *ZustOR*

Gibt den OR-Zustand zurück, der zu dieser Aktion gehört.

\+ setState (*ZustOR*) → *void*

Setzt den OR-Zustand für diese Aktion.

+ execute () → *boolean*
  überschreibt die Methode der Klasse *Aktion*.
  Löscht die History des beim Erschaffen übergebenen Zustands.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert diese Aktion in den EasyFileWriter.
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.3.14 Klasse DeepClearHistoryAktion

extends *score.Aktionen.Aktion*
implements *score.Aktionen.HistoryAktion*

Diese Klasse spezialisiert die abstrakte Klasse Aktion als DeepClearHistoryAktion.

Konstruktoren

+ DeepClearHistoryAktion (*ZustOR*)
  Erschafft eine neue Aktion, die die History des übergebenen Zustands und seiner Unterzustände löscht.

Methoden

+ getState () → *ZustOR*
  Gibt den OR-Zustand zurück, der zu dieser Aktion gehört.
+ setState (*ZustOR*) → *void*
  Setzt den OR-Zustand für diese Aktion.
+ execute () → *boolean*
  überschreibt die Methode der Klasse *Aktion*.
  Löscht die History des beim Erschaffen übergebenen Zustands und seiner Unterzustände.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert die Aktion in den EasyFileWriter.
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.3.15 Klasse ExportAktion

(abstrakte Klasse)
extends *score.Aktionen.AktKomplex*
Diese Klasse erweitert die AktKomplex für den Export. Sie wird lediglich mit einem String aufgerufen, der die zugehörige Komplexe Aktion kennzeichnet: `COMPLEX ACTION nr`. Die Methode execute ist entsprechend überladen.

Attribute

+$ ask:*ConditionAskable* – die Klasse, die Variablen zurückgibt
+$ evHolder:*EventHolder* – die Klasse, in den aufgetretene Ereignisse eingetragen werden
+$ dlgManager:*DialogCompManager* – die Klasse, die den Zugriff auf Dialogkomponenten verwaltet

Konstruktoren

+ ExportAktion (*String*)

Erschafft eine neue ExportAktion mit dem übergebenen Aktions-String.

Methoden

+ execute (*Actor*) → *boolean*

  überschreibt die Methode der Klasse *AktKomplex*.
  Führt diese Aktion aus. Dafür wird die Methode executeCompAction aufgerufen, die vom Quelltextexport geliefert wird.

+ throwEvent (*Object*) → *void*

  Löst das übergebene Event aus, indem es in den EventHolder eingefügt wird. Diese Methode kann bei JAVA-Aktionen benutzt werden.

+ variable (*String*) → *Variable*

  Gibt die Variable mit dem angegebenen Namen zurück.

+ visible (*int*) → *void*

  Macht ein Dialogelement sichtbar.

+ invisible (*int*) → *void*

  Macht ein Dialogelement unsichtbar.

+ on (*int*) → *void*

  Schaltet ein Dialogelement ein.

+ off (*int*) → *void*

  Schaltet ein Dialogelement aus.

executeCompAction (*String*) → *void*

  Diese abstrakte Methode stellt das eigentliche Ausführen einer Aktion parat. Sie wird durch den Quelltextexport entsprechend durch eine `switch`-Anweidung implementiert.

D.3.16 Klasse IncVarAktion

extends *score.Aktionen.SetVarAktion*
Diese Klasse spezialisiert die abstrakte Klasse Aktion als SetVarAktion.

Konstruktoren

+ IncVarAktion (*String, String*)

  Erschafft eine neue Aktion, die eine String-Variable setzt.

+ IncVarAktion (*String, int*)

  Erschafft eine neue Aktion, die eine Integer-Variable setzt.

+ IncVarAktion (*String, double*)

  Erschafft eine neue Aktion, die eine Double-Variable setzt.

+ IncVarAktion (*IncVarAktion*)

  CopyConstructor

Methoden

+ execute () → *boolean*

  überschreibt die Methode der Klasse *SetVarAktion*.
  Setzt den Wert der Variablen.

+ exportQT (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *SetVarAktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.3.17 Klasse JavaAktion

extends *score.Aktionen.Aktion*
Diese Klasse spezialisiert die abstrakte Klasse Aktion als JavaAktion. Sie enthält einen Popup für die Simulation, der auch den Kurznamen dieser Aktion enthält.

Konstruktoren

+ JavaAktion (*String, String, DlgPopup*)
  Erschafft eine neue Aktion mit dem übergebenen Quelltext.

Methoden

+ init (*String, String, DlgPopup*) → *void*
  Initialisiert diese JavaAktion mit einem Namen, Quelltext und einem Popup.
+ delete () → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Löscht diese Aktion und das zugeh;rige Popup.
+ getPopup () → *DlgPopup*
  Gibt das Popup dieser JavaAktion zurück.
+ execute () → *boolean*
  überschreibt die Methode der Klasse *Aktion*.
  Führt prinzipiell den übergebenen Quelltext aus.
+ getName () → *String*
  Gibt den Namen dieser JavaAktion zurück.
+ getCommand () → *String*
  Gibt den auszuführenden Quelltext dieser JavaAktion zurück.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert den Aufruf dieser Aktion in den EasyFileWriter
+ saveBody (*EasyFileWriter*) → *void*
  Speichert den Quelltext zu dieser Aktion in den EasyFileWriter
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert diese Java Aktion. Wenn sie aus dem String `LOAD <filename>` besteht, wird versucht, das File `extern/<filename>.action` an dieser Stelle einzufügen. Schlägt dies fehl, wird stattdessen ein entsprechender Kommentar in das erzeugte File eingetragen.
+$ createFilename (*String, String*) → *String*
  Diese statische Methode erzeugt zu einem Projekt und einem File einen Filename, unter dem eine JavaAktion gespeichert werden kann.
+ createFilename (*String*) → *String*
  Gibt einen Filenamen zurück, unter dem diese JavaAktion gespeichert werden kann.

D.3.18 Klasse SetDialogOnAktion

extends *score.Aktionen.Aktion*
implements *score.Aktionen.DialogAktion*

Diese Klasse spezialisiert die abstrakte Klasse Aktion als SetDialogOnAktion

Konstruktoren

+ SetDialogOnAktion (*DlgKomponente, boolean*)

Erzeugt eine neue Aktion, die die übergebene Dialogkomponente anschaltet.

Methoden

+ execute () → *boolean*
  überschreibt die Methode der Klasse *Aktion*.
  Startet den Timer.
+ setDlgKomponente (*DlgKomponente*) → *void*
  Setzt die Dialogkomponente für diese Aktion.
+ getDlgKomponente () → *DlgKomponente*
  Gibt die DialogKomponente dieser Aktion zurück.
+ setToOn (*boolean*) → *void*
  Setzt die Angabe, ob diese Aktion die DialogKomponente anschaltet.
+ getToOn () → *boolean*
  Gibt zurück, ob diese Aktion die DialogKomponente anschaltet.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert die Aktion im EasyFileWriter.
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.3.19 Klasse SetDialogVisibleAktion

extends *score.Aktionen.Aktion*
implements *score.Aktionen.DialogAktion*

Diese Klasse spezialisiert die abstrakte Klasse Aktion als SetDialogVisibleAktion.

Konstruktoren

+ SetDialogVisibleAktion (*DlgKomponente, boolean*)
  Erschafft eine neue Aktion, die die übergebene Dialogkomponente sichtbar oder unsichtbar werden lässt.

Methoden

+ execute () → *boolean*
  überschreibt die Methode der Klasse *Aktion*.
  Macht die Dialogkomponente sichtbar oder unsichtbar.
+ setDlgKomponente (*DlgKomponente*) → *void*
  Setzt die Dialogkomponente, auf die sich diese Aktion bezieht.
+ getDlgKomponente () → *DlgKomponente*
  Gibt die Dialogkomponente dieser Aktion zurück.
+ setToVisible (*boolean*) → *void*
  Setzt die Angabe, ob diese Aktion die Dialogkomponente sichtbar oder unsichtbar macht.
+ getToVisible () → *boolean*
  Gibt an, ob diese Aktion die Dialogkomponente sichtbar oder unsichtbar macht.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert die Aktion im EasyFileWriter.
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.3.20 Klasse SetVarAktion

extends *score.Aktionen.Aktion*
Diese Klasse spezialisiert die abstrakte Klasse Aktion als SetVarAktion.

Attribute

\# SAVE_KEY:*String* – der String, der beim Speichern als Key für diese Aktion benutzt wird

\# PRINT_KEY:*String* – der String, der beim Anzeigen als Key für diese Aktion benutzt wird

\# name:*String* – der Name der Variable, die zu dieser Aktion gehört

\# sValue:*String* – der String-Wert für die Variable in dieser Aktion

\# dValue:*double* – der Double-Wert für die Variable in dieser Aktion

\# iValue:*int* – der Integer-Wert für die Variable in dieser Aktion

\# mode:*int* – Der Modus gibt an, ob diese Aktion eine String, Double oder Integer-Aktion ist. Die entsprechende Konstante wird aus der Klasse Variable übernommen.

\# setToVar:*boolean* – Gibt an, ob der String-Wert dieser Aktion als Variablenname interpretiert werden soll.

\#$ varserver:*VarServer* – der Variablenserver für alle Aktionen, die sich auf Variablen beziehen

Konstruktoren

\+ SetVarAktion (*String, String*)
Erschafft eine neue Aktion, die eine String-Variable setzt.

\+ SetVarAktion (*String, int*)
Erschafft eine neue Aktion, die eine Integer-Variable setzt.

\+ SetVarAktion (*String, double*)
Erschafft eine neue Aktion, die eine Double-Variable setzt.

\+ SetVarAktion (*SetVarAktion*)
CopyConstructor

Methoden

\+ set (*String, int*) → *void*
Macht diese Aktion zu einer Integer-Aktion mit den übergebenen Spezifikationen.

\+ set (*String, String*) → *void*
Macht diese Aktion zu einer String-Aktion mit den übergebenen Spezifikationen.

\+ set (*String, double*) → *void*
Macht diese Aktion zu einer Double-Aktion mit den übergebenen Spezifikationen.

\+ setName (*String*) → *void*
Setzt den Namen der Variablen.

\+ setCheckVar (*String*) → *void*
Setzt den Namen der Variablen, deren Wert übernommen wird. Alle anderen Angaben werden entsprechend gesetzt, d. h. vorherige Angaben (String, Double, Integer) werden überschrieben.

\+$ setVarServer (*VarServer*) → *void*
Setzt die Simulation, aus der die Aktion die Variable erhält.

\+ setsToVariable () → *boolean*
Sagt dieser Aktion, dass sie die übergebene Variable auf den Wert einer anderen Variablen setzen soll. Dies geht nur dann, wenn sie vorher auf einen String gesetzt hat, in diesem Fall wird der String nun als Variablenname verwendet.

\+ isString () → *boolean*
Gibt zurück, ob es sich um eine String-Variable handelt.

\+ isDouble () → *boolean*

Gibt zurück, ob es sich um eine Double-Variable handelt.

+ isInt () → *boolean*
Gibt zurück, ob es sich um eine Integer-Variable handelt.

+ isToVar () → *boolean*
Gibt an, ob der String-Wert als Variablenname zu interpretieren ist.

+ getVarName () → *String*
Gibt den Namen der zu dieser Aktion gehörenden Variable zurück.

+ getCheckedVarName () → *String*
Gibt den Namen der Source-Variable zurück, falls diese Aktion sich auf zwei Variablen bezieht, ansonsten `Null`.

+ getValueAsString () → *String*
Gibt eine String-Repräsentation des zu setzenden Wertes zurück. Ein String-Wert ist dabei in Anführungszeichen eingefasst; fehlen diese, handelt es sich um einen Variablennamen.

+ execute () → *boolean*
überschreibt die Methode der Klasse *Aktion*.
Setzt den Wert der Variablen.

+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *Aktion*.
Speichert die Aktion im EasyFileWriter.

+ exportQT (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *Aktion*.
Exportiert den Quelltext dieser Aktion in den EasyFileWriter.

D.3.21 Klasse ThrowAktion

extends *score.Aktionen.Aktion*
Diese Klasse spezialisiert die abstrakte Klasse Aktion als ThrowAktion.

Attribute

+$ evHolder:*EventHolder* – Der Vector, in dem alle ThrowAktionen ihre Ereignisse ablegen, wenn sie ausgelöst werden.

Konstruktoren

+ ThrowAktion (*Ereignis*)
Erschafft eine neue Aktion, die das übergebene Ereignis auslöst.

Methoden

+ delete () → *void*
überschreibt die Methode der Klasse *Aktion*.
Löscht diese ThrowAktion. Wenn das Ereignis von keiner anderen ThrowAktion geworfen wird, wird es ebenfalls gelöscht und aus allen Pfeilen ausgetragen.

+ getEvent () → *Ereignis*
Gibt das Ereignis zurück, das von dieser Aktion erzeugt wird.

+ setEvent (*Ereignis*) → *void*
Setzt das Ereignis, das von dieser Aktion erzeugt wird, auf den übergebenen Wert.

+$ getThrowerOf (*Ereignis*) → *Vector*
Gibt die ThrowAktion zurück, die das übergebene Ereignis erzeugt, oder einen leeren Vector, wenn es keine solche gibt.

+ execute () → *boolean*

überschreibt die Methode der Klasse *Aktion*.
Löst das Ereignis aus.

+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Speichert diese Aktion im EasyFileWriter.

+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Aktion*.
  Exportiert den Quelltext für diese Aktion in den EasyFileWriter.

D.4 Package score.Bedingungen

D.4.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.Bedingungen.Bedingung*
    ↪ *score.Bedingungen.BedAtomar*
      ↪ *score.Bedingungen.CompVarBed*
      ↪ *score.Bedingungen.HasHistoryBed*
      ↪ *score.Bedingungen.InStateBed*
      ↪ *score.Bedingungen.JavaBed*
    ↪ *score.Bedingungen.BedKomplex*
      ↪ *score.Bedingungen.BedAND*
      ↪ *score.Bedingungen.BedOR*
    ↪ *score.Bedingungen.BedNOT*
    ↪ *score.Bedingungen.ExportBedingung*

D.4.2 Interface ConditionAskable

Dieses Interface wird von der Klasse implementiert, die es den Bedingungen ermöglicht, ihren Wahrheitswert in der aktuellen Simulation festzustellen.

Methoden

+ getSimBasic (*int*) → *SimBasic*
  Gibt den atomaren Simulationszustand mit dem übergebenen Identifier zurück.

+ getVariable (*String*) → *Variable*
  Gibt die Simulationsvariable mit dem übergebenen Namen zurück.

D.4.3 Klasse BedAND

extends *score.Bedingungen.BedKomplex*
Diese Klasse kombiniert mehrere Bedingungen durch AND.

Konstruktoren

+ BedAND ()
  Erschafft eine neue, leere BedAND.

Methoden

+ isTrue () → *boolean*
  überschreibt die Methode der Klasse *BedKomplex*.
  Gibt an, ob alle Unterbedingungen wahr sind.

+ translate (*Hashtable[]*) → *Bedingung*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt eine Übersetzung dieser AND-Bedingung zurück.

D.4.4 Klasse BedAtomar

(abstrakte Klasse)
extends *score.Bedingungen.Bedingung*
Diese Klasse ist die abstrakte Oberklasse für alle atomaren Bedingungen.

Attribute

+$ ATCOND_JAVA:*int* `CONST` – Konstante, falls es sich um eine Java-Bedingung handelt.

+$ ATCOND_IN:*int* `CONST` – Konstante, falls es sich um eine InState-Bedingung handelt.

+$ ATCOND_HAS_HIST:*int* `CONST` – Konstante, falls es sich um eine HasHistory-Bedingung handelt.

+$ ATCOND_COMP_VAR:*int* `CONST` – Kosntante, falls es sich um eine CompVar-Bedingung handelt.

Konstruktoren

+ BedAtomar (*int*)

Setzt den Typ für die neu erschaffene Bedingung.

Methoden

+ delete () → *void*

überschreibt die Methode der Klasse *Bedingung*.
Löscht diese atomare Bedingung, d. h. sie wird aus dem Vector entfernt.

+ getID () → *int*

Gibt den Identifier dieser Bedingung zurück.

+ getAtomType () → *int*

Gibt den Typ der Bedingung zurück.

+ save (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *Bedingung*.
Speichert die Bedingung im EasyFileWriter. Muss entsprechend implementiert werden.

+ exportAtom (*EasyFileWriter*) → *void*

Exportiert den Quelltext dieser atomaren Bedingung in den EasyFileWriter. Muss entsprechend implementiert werden.

+ exportQT (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *Bedingung*.
Exportiert diese Bedingung in den EasyFileWriter. Diese Methode wird im Zusammenhang mit der `switch`-Anweisung benutzt, die sämtliche (auch komplexen) Bedingungen für die Pfeile exportiert. Für eine atomare Bedingung entspricht dies lediglich einem Aufruf der entsprechenden `switch`-Anweisung der atomaren Bedingungen. Der eigentliche Export geschieht in der Methode exportAtom.

+$ startExport () → *void*

Initialisiert den Export; d. h. es wird zunächst keine Bedingung für den Export markiert.

+$ exportAll (*EasyFileWriter*) → *void*

Exportiert alle Bedingungen, die vorher zum Export markiert wurden indem ihre Methode exportQT aufgerufen wurde.

D.4.5 Klasse Bedingung

extends *java.lang.Object*
Diese Klasse stellt eine Bedingung dar.

Attribute

#$ ask:*ConditionAskable* – Das Interface, bei dem atomare Bedingungen ihren Wahrheitswert erfragen können.

+$ COND_TRUE:*int* CONST – Konstante, falls es sich um die immer wahre Bedingung handelt.

+$ COND_BASIC:*int* CONST – Konstante, falls es sich um eine atomare Bedingung handelt.

+$ COND_AND:*int* CONST – Konstante, falls es sich um eine AND-Bedingung handelt.

+$ COND_OR:*int* CONST – Konstante, falls es sich um eine OR-Bedingung handelt.

+$ COND_NOT:*int* CONST – Konstante, falls es sich um eine NOT-Bedingung handelt.

Konstruktoren

+ Bedingung ()
Erschafft eine immer wahre atomare Bedingung.

Methoden

+ delete () → *void*
Löscht diese Bedingung.

+ getType () → *int*
Gibt den Typ dieser Bedingung zurück. Sollte es sich um eine atomare Bedingung handeln, ist dadurch noch nicht feststellbar, welche atomare Bedingung es ist.

+$ provideAnswers (*ConditionAskable*) → *void*
Setzt die Klasse, die den Wahrheitswert dieser Bedingung liefert.

+ isTrue () → *boolean*
Gibt zurück, ob diese Bedingung erfüllt ist. Muss entsprechend implementiert werden.

+ getCheckedVars (*Vector*) → *void*
Sammelt alle überprüften Variablen im übergebenen Vector.

+ toString () → *String*
überschreibt die Methode der Klasse *Object*.
Gibt diese Bedingung als String zurück. Muss entsprechend implementiert werden und wird für die Darstellung in Listen benötigt.

+ translate (*Hashtable[]*) → *Bedingung*
Gibt eine Übersetzung dieser Bedingung zurück.

+ save (*EasyFileWriter*) → *void*
Speichert die Bedingung im EasyFileWriter.

+ exportQT (*EasyFileWriter*) → *void*
Exportiert den Quelltext für diese Bedingung in den EasyFileWriter.

D.4.6 Klasse BedKomplex

(abstrakte Klasse)
extends *score.Bedingungen.Bedingung*
Erweitert die Bedingung zu einer komplexen Bedingung, d. h. einer, die weitere Unterbedingungen verwalten kann.

Attribute

subConds:*Vector* – Der Vector der Unterbedingungen.

op:*String* – Der Operator als String: AND oder OR.

Konstruktoren

+ BedKomplex (*int*)

Erschafft eine neue komplexe Bedingung ohne Unterbedingungen.

Methoden

+ delete () → *void*
  überschreibt die Methode der Klasse *Bedingung*.
  Löscht diese komplexe Bedingung und alle Unterbedingungen.
+ addCond (*Bedingung*) → *void*
  Fügt eine Unterbedingung hinzu.
+ removeCond (*Bedingung*) → *void*
  Entfernt eine Unterbedingung.
+ getConds () → *Vector*
  Gibt die Unterbedingungen als Vector zurück.
+ isTrue () → *boolean*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt den Wahrheitswert dieser Bedingung zurück. Muss für AND bzw. OR entsprechend implementiert werden.
+ getCheckedVars (*Vector*) → *void*
  überschreibt die Methode der Klasse *Bedingung*.
  Sammelt alle überprüften Variablen im übergebenen Vector.
fillFrom (*BedKomplex, Hashtable[]*) → *void*
  Füllt diese komplexe Bedingung mit den Übersetzungen der übergebenen komplexen Bedingung.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Bedingung*.
  Speichert die Bedingung und ihre Unterbedingungen im EasyFileWriter.
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *Bedingung*.
  Exportiert den Quelltext für diese komplexe Bedingung in den EasyFileWriter. Diese Methode exportiert für atomare Bedingungen keinen Quelltext, sondern lediglich einen weiteren Aufruf.

D.4.7 Klasse BedNOT

extends *score.Bedingungen.Bedingung*
Erschafft eine Bedingung, die die ihr übergebene Bedingung negiert.

Konstruktoren

+ BedNOT ()
  Erschafft eine leere BedNOT.

Methoden

+ delete () → *void*
  überschreibt die Methode der Klasse *Bedingung*.
  Löscht diese Bedingung.
+ setCond (*Bedingung*) → *void*
  Setzt die Unterbedingung, die negiert werden soll.
+ getCond () → *Bedingung*
  Gibt die Unterbedingung zurück.
+ isTrue () → *boolean*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt den negierten Wahrheitswert der Unterbedingung zurück.
+ getCheckedVars (*Vector*) → *void*

überschreibt die Methode der Klasse *Bedingung*.
Sammelt alle überprüften Variablen im übergebenen Vector.

+ translate (*Hashtable[]*) → *Bedingung*
überschreibt die Methode der Klasse *Bedingung*.
Gibt eine Übersetzung dieser NOT-Bedingung zurück.

+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *Bedingung*.
Speichert die Bedingung und ihre Unterbedingung im EasyFileWriter.

+ exportQT (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *Bedingung*.
Exportiert den Quelltext dieser Bedingung in den EasyFileWriter.

D.4.8 Klasse BedOR

extends *score.Bedingungen.BedKomplex*
Erweitert die komplexe Bedingung im Sinne einer OR-Bedingung.

Konstruktoren

+ BedOR ()
Erschafft eine neue, leere BedOR.

Methoden

+ isTrue () → *boolean*
überschreibt die Methode der Klasse *BedKomplex*.
Gibt zurück, ob mindestens eine der Unterbedingungen wahr ist. Die weiteren werden nicht mehr getestet.

+ translate (*Hashtable[]*) → *Bedingung*
überschreibt die Methode der Klasse *Bedingung*.
Gibt eine Übersetzung dieser OR-Bedingung zurück.

D.4.9 Klasse CompVarBed

extends *score.Bedingungen.BedAtomar*
Spezialisiert die atomare Bedingung zu einer Abfrage einer Variablen.

Konstruktoren

+ CompVarBed (*String*, *String*, *int*)
Erschafft eine neue Bedingung, die sich auf die Variable mit dem übergebenen Namen bezieht, mit dem entsprechenden Operator und Vergleichswert.

+ CompVarBed (*String*, *int*, *int*)
Erschafft eine neue Bedingung, die sich auf die Variable mit dem übergebenen Namen bezieht, mit dem entsprechenden Operator und Vergleichswert.

+ CompVarBed (*String*, *double*, *int*)
Erschafft eine neue Bedingung, die sich auf die Variable mit dem übergebenen Namen bezieht, mit dem entsprechenden Operator und Vergleichswert.

Methoden

+ comparesToVariable () → *boolean*
Setzt diese Bedingung auf den Vergleich einer Variablen mit einer Variablen. Dies geht nur bei einer Bedingung, die einen String als zweites Argument übergeben bekam, ansonsten wird `false` zurückgegeben. Der übergebene Vergleichsstring wird bei gültigem Aufruf nun als Variablenname verwendet. Es ist damit allerdings nicht sichergestellt, dass diese Variable existiert - tut sie dies nicht, ergibt jeder Vergleich `false`.

+ getCheckedVars (*Vector*) → *void*

überschreibt die Methode der Klasse *Bedingung*.
Sammelt alle überprüften Variablen im übergebenen Vector.

+ translate (*Hashtable[]*) → *Bedingung*

überschreibt die Methode der Klasse *Bedingung*.
Gibt eine Übersetzung dieser Bedingung zurück.

+ isTrue () → *boolean*

überschreibt die Methode der Klasse *Bedingung*.
Gibt zurück, ob die Vergleichsbedingung erfüllt ist.

+ save (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *BedAtomar*.
Speichert die Bedingung im EasyFileWriter.

+ exportAtom (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *BedAtomar*.
Exportiert den Quelltext für diese atomare Bedingung in den EasyFileWriter.

D.4.10 Klasse ExportBedingung

(abstrakte Klasse)
extends *score.Bedingungen.Bedingung*
Diese abstrakte Klasse bietet die Methoden, die für alle exportierten Bedingungen gemeinsam sind; außerdem hält sie das *ConditionAskable*-Interface, bei dem sie sich nach ihrem Wahrheitswert erkundigen können.

Attribute

+$ ask:*ConditionAskable* – das ConditionAskable Interface, bei dem sich alle atomaren Bedingungen nach ihrem Wahrheitswert erkundigen können.

Konstruktoren

+ ExportBedingung (*int*)

Erschafft eine neue ExportBedingung mit dem angegebenen Identifier.

Methoden

+ isTrue () → *boolean*

überschreibt die Methode der Klasse *Bedingung*.
Gibt zurück, ob diese Bedingung wahr ist.

+ variable (*String*) → *Variable*

Gibt die Variable mit dem angegebenen Namen zurück.

+ in (*int*) → *boolean*

Gibt an, ob der Zustand mit dem übergebenen Identifier gerade aktiv ist.

+ hasHist (*int*) → *boolean*

Gibt an, ob der Zustand mit dem übergebenen Identifier eine History besitzt.

checkArrowCondition (*int*) → *boolean*

Die abstrakte Methode, die eine Bedingung überprüft.

D.4.11 Klasse HasHistoryBed

extends *score.Bedingungen.BedAtomar*
Spezialisiert die atomare Bedingung zu einer Abfrage, ob ein OR-Zustand eine History besizt.

Konstruktoren

+ HasHistoryBed (*ZustOR*)

Erschafft eine neue Bedingung, die sich auf den übergebenen OR-Zustand bezieht.

Methoden

+ isTrue () → *boolean*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt zurück, ob der OR-Zustand eine History gesetzt hat.
+ translate (*Hashtable[]*) → *Bedingung*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt eine Übersetzung dieser Bedingung zurück.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *BedAtomar*.
  Speichert die Bedingung im EasyFileWriter.
+ exportAtom (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *BedAtomar*.
  Exportiert den Quelltext für diese atomare Bedingung in den EasyFileWriter.

D.4.12 Klasse InStateBed

extends *score.Bedingungen.BedAtomar*
Spezialisiert die atomare Bedingung zu einer Anfrage, ob ein bestimmter Zustand momentan aktiv ist.

Konstruktoren

+ InStateBed (*ZustandAtom*)
  Erschafft eine Bedingung, die sich auf den übergebenen Zustand bezieht.

Methoden

+ translate (*Hashtable[]*) → *Bedingung*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt eine Übersetzung dieser Bedingung zurück.
+ isTrue () → *boolean*
  überschreibt die Methode der Klasse *Bedingung*.
  Gibt zurück, ob der Zustand momentan aktiv ist.
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *BedAtomar*.
  Speichert die Bedingung im EasyFileWriter.
+ exportAtom (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *BedAtomar*.
  Exportiert den Quelltext für diese atomare Bedingung in den EasyFileWriter.

D.4.13 Klasse JavaBed

extends *score.Bedingungen.BedAtomar*
Spezialisiert die atomare Bedingung für Java-Quellcode. Sie enthält eine Checkbox für die Simulation, die auch den Kurznamen dieser Bedingung enthält.

Konstruktoren

+ JavaBed (*String, DlgCheckbox*)
  Erschafft eine neue Bedingung, die zum Test den übergebenen Quellcode ausführt.

Methoden

+ delete () → *void*

überschreibt die Methode der Klasse *BedAtomar*.
Löscht diese Bedingung; dabei soll auch die zugehörige Checkbox gelöscht werden.

\+ translate (*Hashtable[]*) → *Bedingung*
überschreibt die Methode der Klasse *Bedingung*.
Gibt diese JavaBed zurück, ohne sie zu kopieren.

– init (*String, String, DlgCheckbox*) → *void*
Initialisiert diese JavaBedingung mit einem Namen, Quellcode und einer Checkbox

\+ isTrue () → *boolean*
überschreibt die Methode der Klasse *Bedingung*.
Gibt prinzipiell zurück, was der Quellcode zurückgibt.

\+ getName () → *String*
Gibt den Namen dieser JavaBed zurück.

\+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *BedAtomar*.
Speichert die Bedingung im EasyFileWriter.

\+ exportAtom (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *BedAtomar*.
Exportiert den Quelltext für diese atomare Bedingung. Hierbei wird hauptsächlich der eingegebene Quelltext in des EasyFileWriter geschrieben.

+$ createFilename (*String, String*) → *String*
Diese statische Methode gibt einen Filename für ein Projekt und ein File zurück, unter dem eine JavaBedingung gespeichert werden kann.

\+ createFilename (*String*) → *String*
Gibt einen Filename zurück, unter dem diese JavaBed gespeichert werden kann.

D.5 Package score.control

D.5.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.control.AbstractParser*
    ↪ *score.control.BedParser*
    ↪ *score.control.Parser*
  ↪ *score.control.CommandPort*
    ↪ *score.control.CommandWindow*
  ↪ *score.control.Commander*
  ↪ *score.control.Interpreter*
    ↪ *score.control.SCInterpreter*
  ↪ *score.control.MainWindow*
  ↪ *score.control.Scanner*
  ↪ *score.control.Token*
  ↪ *score.util.Observable*
    ↪ *score.control.PropertyView*
      ↪ *score.control.Inspector*

D.5.2 Interface Logger

Dies ist ein Interface für Log-Funktionen

Methoden

\+ addLine (*String*) → *void*
Fügt eine Zeile ins Log hinzu

D.5.3 Klasse AbstractParser

(abstrakte Klasse)
extends *java.lang.Object*
Abstrakte Parser Klasse für BedParser und Parser

Attribute

\# log:*Logger* – Ausgabe Logger

\# success:*boolean* – War das Parsen fehlerfrei

Konstruktoren

\+ AbstractParser (*Logger*)
Konstruktor

Methoden

\# error (*String, Token*) → *int*
Gibt eine Fehlermeldung aus

\# error (*String*) → *int*
Gibt eine Fehlermeldung aus

\# checkError (*Token*) → *int*
Prüft ob das Token ein Error-Token war

\+ wasSuccess () → *boolean*
Liefert zurück ob das Parsen erfolgreich war

\# parse () → *int*
Startet das Parsen

D.5.4 Klasse BedParser

extends *score.control.AbstractParser*
Parsed eine Bedingung

Konstruktoren

– BedParser (*Logger*)
Erzeugt einen inneren Parseraufruf

\+ BedParser (*Scanner, Logger*)
Erzeugt einen neuen BedParser

Methoden

\# parse () → *int*
überschreibt die Methode der Klasse *AbstractParser*.
Startet einen Parse-Vorgang

\+ getCondition () → *Bedingung*
Liefert die erzeugte Bedingung

\+$ createCondition (*String*) → *Bedingung*
Erzeugt eine Bedingung aus einem String

D.5.5 Klasse Commander

extends *java.lang.Object*
Diese Klasse wird hauptsächlich für das Laden der Modelle benutzt. Hier kann der Interpreter die gelesenen Daten in einheitlicher Form abgeben, und die entsprechenden Aktionen werden in den Modellen ausgeführt. Außerdem können hier Makros einprogrammiert werden. Auch Speichern und Undo sind über diese Klasse möglich.

Konstruktoren

+ Commander (*MainWindow*)
  Erschafft einen neuen Commander zum übergebenen MainWindow.

Methoden

+ insertCheckboxes (*int*) → *void*
  Erschafft n Checkboxes und fügt sie ein.
+ insertRadiobuttons (*int*) → *void*
  Fügt n Radiobuttons in das DialogModell und das Statechart ein.
+ createBasicState (*int, String, Dimension.Double, boolean*) → *String*
  Erschafft einen atomaren Zustand mit angegebenem Identifier, Name und Größe.
+ createORState (*int, String, Dimension.Double*) → *String*
  Erschafft einen OR-Zustand mit angegebenem Identifier, Name und Größe.
+ createANDState (*int, String, Dimension.Double*) → *String*
  Erschafft einen AND-Zustand mit angegebenem Identifier, Name und Größe.
+ createConnDefEntry (*int, Dimension.Double*) → *String*
  Erschafft einen DefaultEntryKonnektor mit angegebenem Identifier und Größe.
+ createConnUniversal (*int, String, Dimension.Double*) → *String*
  Erschafft einen Universalkonnektor mit angegebenem Identifier, Name und Größe.
+ createConnHistory (*int, String, Dimension.Double, boolean*) → *String*
  Erschafft einen HistoryKonnektor, mit angegebenem Identifier, Name, Größe und Tiefe.
+ createText (*int, Point2D.Double, String*) → *String*
  Erschafft einen neuen GrafText mit angegebenem Identifier, Offset und Text.
+ insert (*int, int, Point2D.Double*) → *String*
  Fügt einen Zustand oder Konnektor in einen Oberzustand an die angegebene Stelle ein.
+ addView (*int, int, double, boolean, boolean, boolean, boolean*) → *String*
  Fügt zum angegebenen Zustand eine Sicht mit entsprechender Tiefe, Zoom und Darstellung hinzu.
+ addPartitioner (*int, Point2D.Double, Point2D.Double*) → *String*
  Fügt zum angegebenen AND-Zustand einen Partitioner zwischen den angegebenen Punkten hinzu.
+ addText (*int, int*) → *String*
  Fügt einen GrafText zum angegebenen Zustand hinzu.
+ createArrow1_1 (*int, Vector*) → *String*
  Erschafft einen 1-1 Pfeil mit angegebenen Identifier und Hook-Vector.
+ createArrow1_N (*int, Vector, Triplet, Vector*) → *String*
  Erschafft einen 1-N Pfeil mit angegebenen Identifier, Hookvector für den Startabschnitt, KonnektorHook und Endabschnitten aus dem angegebenen Vector.
+ createArrowN_1 (*int, Vector, Triplet, Vector*) → *String*
  Erschafft einen N-1 Pfeil mit angegebenem Identifier, Startabschnitten aus dem angegebenen Vector, KonnektorHook und Hookvector für den Endabschnitt.
– parseHook (*GrafPfeil, Triplet*) → *GrafHook*

Übersetzt das Triplet in einen Hook. Format: ID des GrafPunkt-Anchors, Position, TextID

+ createEvent (*int, String*) → *String*
  Erschafft ein neues Ereignis mit dem angegebenen Identifier und Text.
+ createActionClearHist (*int, int*) → *String*
  Erschafft eine neue ClearHistoryAktion mit dem angegebenen Identifier zum angegebenen Zustand.
+ createActionDeepClear (*int, int*) → *String*
  Erschafft eine neue DeepClearHistoryAktion mit dem angegebenen Identifier zum angegebenen Zustand.
+ createActionSetVisible (*int, int, boolean*) → *String*
  Erschafft eine neue SetVisibleAktion mit dem angegebenen Identifier zum angegebenen Dialogelement.
+ createActionDialogOn (*int, int, boolean*) → *String*
  Erschafft eine neue SetDialogOnAktion mit dem angegebenen Identifier zur angegebenen Dialogkomponente.
+ createActionSetVarVar (*int, String, String, boolean*) → *String*
  Erschafft eine neue SetVariableAktion mit dem angegebenen Identifier zur angegebenen String-Variable.
+ createActionSetVar (*int, String, String, boolean*) → *String*
  Erschafft eine neue SetVariableAktion mit dem angegebenen Identifier zur angegebenen String-Variable.
+ createActionSetVar (*int, String, int, boolean*) → *String*
  Erschafft eine neue SetVariableAktion mit dem angegebenen Identifier zur angegebenen Integer-Variable.
+ createActionSetVar (*int, String, double, boolean*) → *String*
  Erschafft eine neue SetVariableAktion mit dem angegebenen Identifier zur angegebenen Double-Variable.
+ createActionThrow (*int, int*) → *String*
  Erschafft eine neue ThrowAktion mit dem angegebenen Identifier zum angegebenen Ereignis.
+ createActionJava (*int, int, String, String*) → *String*
  Erschafft eine neue JavaAktion mit dem angegebenen Identifier mit dem angegebenen Quelltext.
+ createDialog (*DlgKomponente, int, boolean, boolean*) → *String*
  Erschafft ein neues Dialogelement mit den übergebenen Angaben zu Typ, Sichtbarkeit zu Beginn und Zugehörigkeit zum Benutzer- oder Systemdialog.
+ addActionActivate (*int, int*) → *String*
  Fügt die angegebene Aktion zum angegebenen Zustand beim Aktivieren hinzu.
+ addActionDeactivate (*int, int*) → *String*
  Fügt die angegebene Aktion zum angegebenen Zustand beim Deaktivieren hinzu.
+ addActionArrow (*int, int*) → *String*
  Fügt die angegebene Aktion zum angegebenen Pfeil hinzu.
+ addActionDialog (*int, int*) → *String*
  Fügt die angegebene Aktion zum angegebenen Dialogelement hinzu.
+ addCondition (*int, Bedingung*) → *String*
  Fügt die übergebene Bedingung zum angegebenen Pfeil hinzu.
+ addEvent (*int, int*) → *String*
  Setzt das übergebene Ereignis als Trigger des angegebenen Pfeils.
+ save (*String*) → *String*

Speichert das logische und grafische Statechart Modell unter dem angegebenen Dateinamen.

+ store () → *String*

  Speichert das Statechart in einen String, der im MainWindow auf den undoStack gelegt wird.

+ endLoad () → *String*

  schließt den Ladevorgang ab

+ exportQT (*String*) → *String*

  Exportiert den Quelltext des Statecharts.

+ enableRefresh (*boolean*) → *void*

  Ermöglicht oder verhindert das Refresh der Sichten.

+ refresh () → *String*

  Führt ein Refresh der Sichten aus.

+ clearAll () → *String*

  Löscht sämtliche Informationen des Editors.

+ prepareForUndo () → *void*

  Bereitet alles für ein Undo vor.

+ finishUndo () → *void*

  Beendet einen Undo-Vorgang.

+ getUndoString (*UndoStack*) → *String*

  Gibt das letzte zum Undo gespeicherte Statechart als String zurück. Dieser String entspricht den Befehlen beim Speichern auf Festplatte.

D.5.6 Klasse CommandPort

extends *java.lang.Object*
implements *score.control.Logger*

Diese Klasse stellt die Schnittstelle für Kommandoaktionen dar

Attribute

com:*Commander* – Referenz auf einen Commander

project:*File* – Das aktuelle Projekt

parser:*Parser* – Referenz auf einen Parser

Konstruktoren

+ CommandPort (*File, Commander*)

  Konstruktor

Methoden

init () → *void*

  Erzeugt einen neuen Parser

+ execute (*String*) → *boolean*

  Führt einen String aus (d. h. parsed und interpretiert ihn)

+ addLine (*String*) → *void*

  Logger

D.5.7 Klasse CommandWindow

extends *score.control.CommandPort*
implements *java.awt.event.ActionListener*

Stellt ein Fenster zur direkten Eingabe von Kommandos dar

Konstruktoren

+ CommandWindow (*File, Commander*)
  Konstruktor

Methoden

init () → *void*
  überschreibt die Methode der Klasse *CommandPort*.
  Buttons und Textfelder initialisieren

+ dispose () → *void*
  Den frame löschen

+ execute (*String*) → *boolean*
  überschreibt die Methode der Klasse *CommandPort*.
  Den String s parsen und interpretieren

+ addLine (*String*) → *void*
  überschreibt die Methode der Klasse *CommandPort*.
  Logger: Eine Zeile ins Log

+ actionPerformed (*ActionEvent*) → *void*
  Auf die Buttons reagieren

D.5.8 Klasse Inspector

extends *score.control.PropertyView*
Stellt den eigentlichen Inspektor dar

Konstruktoren

+ Inspector ()
  Konstruktor

+ Inspector (*JFrame*)
  Konstruktor

Methoden

+ redraw () → *void*
  überschreibt die Methode der Klasse *PropertyView*.
  Alles neuzeichnen

+ dispose (*boolean*) → *void*
  überschreibt die Methode der Klasse *PropertyView*.
  Fenster löschen

D.5.9 Klasse Interpreter

(abstrakte Klasse)
extends *java.lang.Object*
implements *score.control.Logger*

Die abstrakte Oberklasse des Interpreter

Attribute

\# parser:*Parser* – Eine Referenz auf den parser

\# logger:*Logger* – Eine Referenz auf den Interpreter

Konstruktoren

\+ Interpreter (*Logger, Parser*)
Erzeugt einen neuen Interpreter

Methoden

\+ interprete (*int*) → *int*
Startet einen Interpretiervorgang

\+ addLine (*String*) → *void*
Logger: Fügt eine neue Zeile ins Log

D.5.10 Klasse MainWindow

extends *java.lang.Object*
implements *java.awt.event.ActionListener,*
score.util.UndoManager,
java.awt.event.WindowListener

MainWindow Hauptfenster mit Toolbar und Menü

Attribute

\+$ undoStack:*UndoStack* – UndoStack

\+$ redoStack:*UndoStack* – RedoStack

\+$ gscm:*GrafSCModell* – Grafisches Statechartmodell

\+$ ldm:*LogDModell* – Logisches Statechartmodell

\+$ simulation:*Simulation* – Der Simulator

\+$ dEditor:*DialogEditor* – Der DialogEditor

\+$ aDialog:*AktionDialog* – Der AktionenDialog

\+$ eDialog:*EventDialog* – Der EreignisDialog

– mainFrame:*JFrame* – Frame für das Hauptfenster

Konstruktoren

\+ MainWindow (*String[]*)
Erstellt das Hauptfenster

Methoden

– quit () → *boolean*
Beenden des Programms. quit prüft zur Zeit noch keine Sicherheitsabfrage

\+ newView (*int*) → *void*
Erzeugt eine neue Ansicht des Statecharts

\+ newView (*int, int*) → *void*
Erzeugt eine neue Ansicht des Statecharts mit Tiefe depth

\+ deleteView (*SCEditor*) → *void*
Löscht die Ansicht v des Statecharts

\+ actionPerformed (*ActionEvent*) → *void*

Funktion zum Event-Handling. implementiert für ActionListener.
Parameter:

- e: Der übergebene Event

– initMenuBar () → *void*
Rückgabewert: Die erstellte MenuBar
Erstellt die MenuBar

\+ commit () → *void*
Bestätigt einen Undo Vorgang

\+ prepare () → *void*
Bereitet einen Undo-Vorgang vor

\+ isPrepared () → *boolean*
Ist ein Undo-Vorgang vorbereitet?

\+ undo () → *void*
Führt ein Undo durch

\+ clearForUndo () → *void*
Bereitet die Modelle für ein Undo vor

\+ finishUndo () → *void*
Beendet einen Undo-Vorgang

\+ initAll () → *void*
Initialisiert das ganze System

\+$ enableToolbars (*boolean*) → *void*
Enabled die Toolbars aller Editoren

\+ windowClosing (*WindowEvent*) → *void*
Lenkt auf die Methode quit um.

D.5.11 Klasse Parser

extends *score.control.AbstractParser*
Parser für die SC Kommandos

Konstruktoren

\+ Parser (*File, Logger, Commander, JProgressBar*)
Erzeugt einen neuen Parser mit Progressbar

\+ Parser (*File, Logger, Commander*)
Erzeugt einen neuen Parser

Methoden

\# parse () → *int*
überschreibt die Methode der Klasse *AbstractParser*.
Startet den Parse-Vorgang

\+ parseCommand (*String, boolean*) → *int*
parsed ein Commando command. Wenn printSummary true ist gibt er auf dem Logger eine Zusammenfassung aus.

D.5.12 Klasse PropertyView

extends *score.util.Observable*
implements *score.util.Observer,*
java.awt.event.ActionListener,
java.awt.event.WindowListener

Stellt die Attribute eines MarkableTyp dar. Wird vom Inspektor benutzt

Attribute

+ frame:*JFrame* – Der Frame in dem der PropertyView dargestellt wird
# font:*Font* – Der Font für den Inspektor
# tabPane:*JTabbedPane* – Die TabbedPane für die markierten Objekte
# observable:*Observable* – Das zu überwachende Observable
# noUpdate:*boolean* – true, wenn kein Update durchgeführt werden soll
# shownHash:*Hashtable* – Hashtable für die dargestellten Objekte

Konstruktoren

+ PropertyView (*Observable*)
Konstruktor
+ PropertyView (*Observable, JFrame*)
Konstruktor
PropertyView ()
Konstuktor

Methoden

+$ setUndoManager (*UndoManager*) → *void*
Setzt den UndoManager
+ replaceObservable (*Observable, Observable*) → *boolean*
Methode die beim Ersetzen des Obervables aufgerufen wird
initRedraw () → *void*
Ein Neuzeichnen vorbereiten, d. h. die Ressourcen anfordern
redrawPane () → *void*
Die TabbedPane darstellen
saveChangedFields () → *void*
Textfelder auslesen und sie Werte speichern
+ redraw () → *void*
Alles neuzeichnen
showField (*MarkableTyp*) → *void*
Felder zeichnen
showFields (*Variable*) → *void*
Felder zeichnen
showFields (*Partitioner*) → *void*
Felder zeichnen
showFields (*DlgKomponente*) → *void*
Felder zeichnen
showFields (*GrafText*) → *void*
Felder zeichnen
showFields (*GrafPfeil*) → *void*

Felder zeichnen

\# showFields (*GrafHook*) → *void*
Felder zeichnen

\# showFields (*GrafPunkt*) → *void*
Felder zeichnen

\+ dispose (*boolean*) → *void*
Dieses Fenster löschen

\+ update (*Observable, Object*) → *void*
Neuzeichnen wenn sich das Modell geändert hat

\+ actionPerformed (*ActionEvent*) → *void*
Werte nach einem Event setzen

D.5.13 Klasse Scanner

extends *java.lang.Object*
Der Scanner unterteilt einen Strint in Tokens

Konstruktoren

\+ Scanner (*String*)
Erzeugt einen neuen Scanner aus einem String

Methoden

\+ nextToken () → *Token*
Liefert das nächste Token

\+ hasMoreTokens () → *boolean*
True wenn es keine Tokens mehr gibt

\+ setParseDims (*boolean*) → *void*
Setzt ob im ParseString Dimensions [] vorkommen können

\+ setParsePoints (*boolean*) → *void*
Setzt ob im ParseString Points vorkommen können

\+ setIgnoreParens (*boolean*) → *void*
Setzt ob Klammern ignoriert werden sollen

D.5.14 Klasse SCInterpreter

extends *score.control.Interpreter*
Der Interpreter für die Statechart-Befehle

Konstruktoren

\+ SCInterpreter (*File, Commander, Logger, Parser, JProgressBar*)
Erzeugt einen neuen Interpreter

Methoden

\+ interprete (*int*) → *int*
überschreibt die Methode der Klasse *Interpreter*.
Interpretiert ein Kommando

D.5.15 Klasse Token

extends *java.lang.Object*
Klasse die ein Token während das Parsens repräsentiert

Konstruktoren

+ Token (*int*)
  Konstruktor
+ Token (*int, boolean*)
  Konstruktor
+ Token (*int, int*)
  Konstruktor
+ Token (*int, double*)
  Konstruktor
+ Token (*int, Point2D.Double*)
  Konstruktor
+ Token (*int, Dimension.Double*)
  Konstruktor
+ Token (*int, String*)
  Konstruktor

Methoden

+ getType () → *int*
  Liefert den Typ des Tokens
+ toString () → *String*
  überschreibt die Methode der Klasse *Object*.
  Liefert einen String des Tokens

D.6 Package score.DEditor

D.6.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *java.awt.event.MouseAdapter*
    ↪ *score.DEditor.DialogEditor*
  ↪ *score.util.Observable*
    ↪ *score.DEditor.DlgKompSelektor*

D.6.2 Klasse DialogEditor

extends *java.awt.event.MouseAdapter*
implements *score.util.Observer,*
java.awt.event.ActionListener,
java.awt.event.WindowListener

Erstellt ein Fenster in dem das Erstellen neuer Dialogkomponenten und das Löschen ermöglicht wird

Attribute

+$ undo:*UndoManager* – Referenz auf den UndoManager

Konstruktoren

+ DialogEditor (*LogDModell*)

Konstruktor
Parameter:

- modell: das logische Dialog Modell

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*
  Ersetzt das Observable
+ actionPerformed (*ActionEvent*) → *void*
  Implementation für ActionListener-Interface
+ update (*Observable, Object*) → *void*
  Implementation für Observer-Interface
+ mousePressed (*MouseEvent*) → *void*
  überschreibt die Methode der Klasse *MouseAdapter*.
  Implementation für MouseListener

D.6.3 Klasse DlgKompSelektor

extends *score.util.Observable*
implements *java.awt.event.ActionListener,*
javax.swing.event.ListSelectionListener

Stellt eine Auswahlbox in der alle Dialogkomponenten oder nur Dialogkomponenten eines Typs dargestellt oder ausgewählt werden können.

Attribute

frame:*JFrame* – Referenz auf den Frame in dem der Selektor dargestellt wird

Konstruktoren

+ DlgKompSelektor (*JFrame, LogDModell, int*)
  Konstruktor
+ DlgKompSelektor (*JFrame, LogDModell*)
  Konstruktor

Methoden

+ makePanel () → *JPanel*
  Erzeugt das Panel des Selektors
+ actionPerformed (*ActionEvent*) → *void*
  Auf ButtonEvents reagieren
+ valueChanged (*ListSelectionEvent*) → *void*
  Wenn sich die ListenSelektion geändert hat, die Buttons updaten
+ setValue (*DlgKomponente*) → *void*
  Die DlgKomponente value in der Liste markieren

D.7 Package score.doclet

D.7.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *com.sun.javadoc.Doclet*
    ↪ *score.doclet.LaTeXDoclet*

D.7.2 Klasse LaTeXDoclet

extends *com.sun.javadoc.Doclet*
Ein Doclet für die LaTeX-Ausgabe. Formatiert die von JavaDoc eingelesenen Daten und gibt sie als LaTeX-Quelltext an die Standardausgabe weiter.

Attribute

–$ nl:*String* `CONST` – Portabler NewLine String

–$ allPrivates:*boolean* – `true`, falls auch unkommentierte, private Attribute und Methoden ausgegeben werden sollen.

Methoden

–$ outItem (*ProgramElementDoc*) → *void*
Beginnt ein neues Item mit der passenden Kombination aus [public, private, protected, package-private], static, final, abstract.
Parameter:
– doc: das ProgrammElementDoc, auf das Bezug genommen wird

–$ outSignature (*ExecutableMemberDoc*) → *boolean*
Rückgabewert: `true`, wenn die Signatur nicht leer ist
Gibt die Signatur einer Methode (oder Konstruktors) aus. Der Inhalt der Klammern wird als Typ gesetzt.
Parameter:
– exm: das ExecutableMemberDoc, dessen Signatur gemeint ist

+$ start (*RootDoc*) → *boolean*
Beginnt die Ausgabe der Dokumentation.
Parameter:
– root: das *RootDoc*, das alle Informationen enthält, die dokumentiert werden sollen

•$ printPackages (*PackageDoc[]*) → *void*
Gibt die Packages eines Arrays aus. Dadurch wird eine neue `section` begonnen. Die Reihenfolge der Klassen und Interfaces ist: Error, Exception, Interface, Klasse.
Parameter:
– packages: der auszugebende Array von PackageDocs

–$ loadHierarchy (*ClassDoc[]*) → *void*
Baut eine Klassenhierarchie der im Array übergebenen Klassen auf.
Parameter:
– classes: der Array der Klassen

–$ printHierarchy () → *void*
Gibt die vorher aufgebaute Klassenhierarchie aus. Dies erzeugt eine eigene `subsection`, die aus geschachtelten `itemize`-Umgebungen bestehen.

•$ printClasses (*ClassDoc[]*) → *void*
Gibt einen Array von Klassendokumentationen aus. Dadurch wird eine neue `subsection` begonnen. Die Reihenfolge der Angaben ist: Attribute, Konstruktoren, Methoden. Weitere Informationen werden am Anfang ausgegeben.
Parameter:
– classes: der auszugebende Array von ClassDocs

•$ printFields (*FieldDoc[]*) → *void*
Gibt die Attribute (Felder) eines Arrays aus. Der Abschnitt wird durch `fkmhead` begonnen und besteht aus einer Itemize-Umgebung.

Parameter:

- fields: der auszugebende Array von FieldDocs

•$ printConstructors (*ConstructorDoc[], boolean*) → *void*
Gibt einen Array von Konstruktordokumentationen aus. Der Abschnitt wird durch `fkmhead` begonnen und besteht aus einer Itemize-Umgebung.
Parameter:

- constructors: der auszugebende Array von ConstructorDocs

•$ printMethods (*MethodDoc[]*) → *void*
Gibt einen Array von Methodendokumentationen aus. Der Abschnitt wird durch `fkmhead` begonnen und besteht aus einer Itemize-Umgebung.
Parameter:

- methods: der auszugebende Array von MethodDocs

•$ printMember (*ExecutableMemberDoc*) → *void*
Gibt die Informationen einer Methode (oder eines Konstruktors) aus. Dazu zählen der Kommentar, die Exceptions, die geworfen werden können, und die Parameterliste.
Parameter:

- mem: das ExecutableMemberDoc, auf das Bezug genommen wird

D.8 Package score.Ereignisse

D.8.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.LogSCM.LogComponent*
    ↪ *score.Ereignisse.Ereignis*
  ↪ *score.util.Observable*
    ↪ *score.Ereignisse.EreignisEdit*
    ↪ *score.Ereignisse.InlineEventDialog*
      ↪ *score.Ereignisse.EventDialog*

D.8.2 Interface EventAskable

Dieses Interface wird von der Klasse implementiert, die den Ereignissen sagt, ob sie gerade eingetreten sind.

Methoden

\+ occurred (*Ereignis*) → *boolean*
Gibt `true` zurück, wenn dieses Ereignis gerade eingetreten ist.

D.8.3 Klasse Ereignis

extends *score.LogSCM.LogComponent*
implements *java.lang.Comparable*

Diese Klasse stellt ein Ereignis dar.

Attribute

+$ ask:*EventAskable* – Das Interface, bei dem Ereignisse anfragen können, ob sie gerade eingetreten sind.

Konstruktoren

\+ Ereignis (*String*)

Erschafft ein neues Ereignis mit dem angegebenen Namen.

Methoden

+ delete () → *void*
überschreibt die Methode der Klasse *Observable*.
Löscht dieses Ereignis aus dem Hashtable aller Ereignisse und löst ein Refresh aus.

+$ getAll () → *Vector*
Gibt einen Vector aller Ereignisse zurück.

+ getID () → *int*
Gibt den Identifier dieses Ereignisses zurück.

+ compareTo (*Object*) → *int*
Implementiert das Comparable-Interface.

+$ getEvent (*int*) → *Ereignis*
Gibt das Ereignis mit dem angegebene Identifiert zurück.

+ setName (*String*) → *void*
Setzt den Namen dieses Ereignisses.

+ getName () → *String*
Gibt den Namen dieses Ereignisses zurück.

+$ provideAnswers (*EventAskable*) → *void*
Setzt die Klasse, die angibt, ob dieses Ereignis eingetreten ist.

+ occurred () → *boolean*
Gibt an, ob dieses Ereignis eingetreten ist.

+ save (*EasyFileWriter*) → *void*
Speichert dieses Ereignis im EasyFileWriter.

D.8.4 Klasse EreignisEdit

extends *score.util.Observable*
implements *java.awt.event.ActionListener*

Stellt ein Fenster dar in dem der String eines Ereignisses editiert werden kann

Konstruktoren

+ EreignisEdit (*Ereignis*)
Konstruktor

Methoden

+ actionPerformed (*ActionEvent*) → *void*
Wertet Knopfdrücke aus

D.8.5 Klasse EventDialog

extends *score.Ereignisse.InlineEventDialog*
implements *java.awt.event.WindowListener*

Stellt ein Fenster mit einem InlineEventDialog dar

Konstruktoren

+ EventDialog ()
Konstruktor

D.8.6 Klasse InlineEventDialog

extends *score.util.Observable*
implements *java.awt.event.ActionListener,*
score.util.Observer,
javax.swing.event.ListSelectionListener

Stellt eine Liste aller Ereignisse dar. Neue können hinzugefügt werden, alte editiert oder gelöscht

Attribute

\# frame:*JFrame* – Referenz auf den Frame in dem der InlineEventDialog dargestellt wird

Konstruktoren

\+ InlineEventDialog (*JFrame*)
Konstruktor

Methoden

\+ replaceObservable (*Observable, Observable*) → *boolean*
Wenn da Modell ersetzt wird, wird diese Funktion implizit aufgerufen

\+ makePanel () → *JPanel*
Erzeugt das Panel des Dialogs

\+ actionPerformed (*ActionEvent*) → *void*
Wertet ButtonEvents aus

\+ update (*Observable, Object*) → *void*
Observerfunktion, wenn sich die Ereignisse ändern

\+ valueChanged (*ListSelectionEvent*) → *void*
Wenn sich die Selektion der Liste geändert hat

\+ setValue (*Ereignis*) → *void*
Setzt die Liste von aussen

D.9 Package score.GrafSCM

D.9.1 Klassenhierarchie

· *java.lang.Object*
↪ *java.util.Vector*
↪ *score.GrafSCM.GPAbschnitt*
↪ *score.AbstractModell.AbstractModell*
↪ *score.GrafSCM.GrafSCModell*
↪ *score.AbstractModell.MarkableTyp*
↪ *score.GrafSCM.GrafTyp*
↪ *score.GrafSCM.GrafHook*
↪ *score.GrafSCM.GrafPfeil*
↪ *score.GrafSCM.GrafPunkt*
↪ *score.GrafSCM.GrafKonnektor*
↪ *score.GrafSCM.GrafZustand*
↪ *score.GrafSCM.GrafAND*
↪ *score.GrafSCM.GrafTopLevelZust*
↪ *score.GrafSCM.GrafText*
↪ *score.GrafSCM.Partitioner*
↪ *score.GrafSCM.GrafModellSicht.visHook*
↪ *score.GrafSCM.GrafModellSicht.visText*
↪ *score.GrafSCM.SuperList*
↪ *score.GrafSCM.Unterpunkt*

↪ *score.util.Observable*
↪ *score.GrafSCM.GrafModellSicht*

D.9.2 Error InvalidConstructionError

extends *java.lang.Error*
Dieser Error wird geworfen, wenn beim Einfügen eines neuen Zustandes ungültige Koordinaten angegeben werden.

Konstruktoren

+ InvalidConstructionError (*String*)
  Erzeugt einen neuen Error mit dem angegebenen Text.

D.9.3 Klasse GPAbschnitt

extends *java.util.Vector*
Diese Klasse stellt einen Abschnitt eines GrafPfeils dar. Sie ist ein Vector von Hooks. Sie kann sowohl ein Anfangs- als auch ein Endabschnitt sein.

Attribute

+$ DROP_NONE:*int* – Konstante, die angibt, dass beim Speichern kein Punkt ausgelassen werden soll

+$ DROP_FIRST:*int* – Konstante, die angibt, dass beim Speichern der erste Punkt ausgelassen werden soll, weil es sich um den KonnektorHook des Pfeils handelt.

+$ DROP_LAST:*int* – Konstante, die angibt, dass beim Speichern der erste Punkt ausgelassen werden soll, weil es sich um den KonnektorHook des Pfeils handelt.

Konstruktoren

+ GPAbschnitt (*GrafPfeil*)
  Erschafft einen neuen Abschnitt zum übergebenen Pfeil, der noch keine Hooks enthält.

+ GPAbschnitt (*GrafPunkt, GrafPunkt, GrafPfeil, Point2D.Double, Point2D.Double*)
  Erschafft einen Abschnitt mit gegebenem Start- und Endpunkt zum übergebenen Pfeil.

+ GPAbschnitt (*GrafPunkt, GrafHook, GrafPfeil, Point2D.Double, Point2D.Double*)
  Erschafft einen Abschnitt mit gegebenem Startpunkt und Endhook zum übergebenen Pfeil.

+ GPAbschnitt (*GrafHook, GrafPunkt, GrafPfeil, Point2D.Double, Point2D.Double*)
  Erschafft einen Abschnitt mit gegebenem Starthook und Endpunkt zum übergebenen Pfeil.

Methoden

– getInitialPos (*GrafPunkt, GrafTyp, Point2D.Double, Point2D.Double*) → *Point2D.Double*
  Gibt die Position zurück, den ein Hook auf dem übergebenen Punkt haben muss, wenn der Pfeil durch klicken an den übergebenen Positionen in den übergebenen Punkten bestimmt wurde. Der zweite Punkt darf auch ein GrafHook sein.

+ getArrow () → *GrafPfeil*
  Gibt den Pfeil zurück, zu dem dieser Abschnitt gehört.

+ splitAt (*GrafHook*) → *GPAbschnitt*
  Spaltet den Abschnitt am gegebenen Hook auf und gibt das neue Endstück zurück. Liegt der übergebene Hook nicht in diesem Abschnitt, wird `Null` zurückgegeben.

+ doubleStart () → *void*
  Verdoppelt den ersten Hook.

+ doubleMid (*GrafHook*) → *void*

Verdoppelt einen mittleren Hook.

+ doubleEnd () → *void*
  Verdoppelt den letzten Hook.
– newHalfHook (*GrafHook, GrafHook*) → *GrafHook*
  Erschafft einen Hook, der genau zwischen den beiden übergebenen Hooks liegt.
– newLowerHook (*GrafHook, GrafHook*) → *GrafHook*
  Erschafft einen Hook, der die Strecke zwischen den beiden übergebenen Hooks im Verhältnis 1:2 teilt.
– getNewHook (*GrafHook, GrafHook, double*) → *GrafHook*
  Erschafft einen neuen Hook mit den angegebenen Teilverhältnis der Zwischenstrecke. Der Hook wird in den entsprechenden Zustand eingefügt, wobei die SuperList des Pfeils benutzt aber nicht verändert wird.
+ relocalize () → *void*
  Stellt sicher, dass sich jeder Hook im korrekten Zustand befindet. Dabei wird die übergebene SuperList benutzt, aber nicht verändert.
+ getCoordsOfSecond (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des zweiten Punktes zurück.
+ getCoordsOfPenultimate (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des vorletzten Punktes zurück.
+ getCoordsOfFirst (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des ersten Punktes zurück.
+ getCoordsOfLast (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des letzten Punktes zurück.
+ getPosOfNeighbours (*GrafHook, GrafModellSicht*) → *Vector*
  Gibt die Zeichenkoordinaten sämtlicher Nachbarzustände zurück.
+ addPosOfCrossing (*Vector, GrafPunkt, GrafModellSicht*) → *void*
  Fügt zum übergebenen Vector die Zeichenkoordinaten der Hooks hinzu, deren Verbindungslinie die Grenzen des übergebenen GrafPunkts schneidet.
+ addHooksJustOutside (*Vector, GrafPunkt*) → *void*
  Fügt zum übergebenen Vector die Hooks dieses Abschnittes hinzu, die gerade außerhalb des übergebene GrafPunkts liegen.
+ hasHookOutside (*GrafPunkt*) → *boolean*
  Gibt an, ob dieser Abschnitt Hooks außerhalb des übergebenen GrafPunkt hat.
+ translate (*Hashtable[], int, GrafPfeil*) → *GPAbschnitt*
  Gibt eine Übersetzung dieses Abschnitts zurück, die zum übergebenen Pfeil gehört. Mit dem Modus wird übergeben, ob der erste oder letzte Hook ein ConnectorHook ist und deshalb nicht mit eingefügt werden soll.
+ save (*EasyFileWriter, int*) → *void*
  Speichert diesen Abschnitt im EasyFileWriter. Der Modus gibt dabei an, ob einer der Endpunkte dabei vernachlässigt werden soll, weil es sich um den KonnektorHook des Pfeils handelt, der gesondert gespeichert wird.
+ save (*EasyFileWriter*) → *void*
  Speichert diesen Abschnitt im EasyFileWriter.
+ getFirstAnchor () → *GrafPunkt*
  Gibt den Anker des ersten Hooks zurück.
+ getLastAnchor () → *GrafPunkt*
  Gibt den Anker des letzten Hooks zurück.
+ delete () → *void*
  Löscht diesen Abschnitt. Die zugehörigen Hooks werden aus ihren Ankerpunkten entfernt, aber ihre delete() Methode wird nicht aufgerufen.

D.9.4 Klasse GrafAND

extends *score.GrafSCM.GrafZustand*
Diese Klasse stellt das grafische Äquivalent eines logischen AND-Zustands dar. Die Unterpunkte sind hierbei, wie im logischen Statechart-Modell, die Partitionen dieses Zustands.

Konstruktoren

+ GrafAND (*Dimension.Double*)
  Erschafft einen grafischen AND-Zustand, der die angegebene Größe hat. Er besitzt anfangs genau eine Partition und noch keinen Partitioner.
+ GrafAND (*Dimension.Double, ZustAND*)
  Erschafft einen neuen grafischen AND-Zustand und initialisiert ihn mit dem übergebenen logischen AND-Zustand.

Methoden

+ isValidSize (*Dimension.Double*) → *boolean*
  überschreibt die Methode der Klasse *GrafZustand*.
  Testet, ob die übergebene Größe für diesen Zustand gültig wäre. Dabei werden Partitionen und Partitioner beachtet.
+ setSize (*Dimension.Double*) → *void*
  überschreibt die Methode der Klasse *GrafPunkt*.
  Setzt die Größe dieses Zustands. Dabei werden Partitionern und Partitioner beachtet.
+ addSubPoint (*GrafPunkt, Point2D.Double*) → *Unterpunkt*
  überschreibt die Methode der Klasse *GrafZustand*.
  Fügt eine Partition hinzu, die bereits eine logische Entsprechung haben muss, die ein OR-Zustand ist. Der neue Punkt hat bezüglich dieses Zustands den angegebenen Offset. Ist dieser `Null`, wird `false` zurückgegeben. Der entsprechende Vorgang wird im logischen Modell wiederholt, falls dieser Zustand eine logische Entsprechung hat, die ein AND-Zustand ist. In diesem Fall wird der neue Unterpunkt zurückgegeben, ansonsten `Null`.
+ addSubRelative (*GrafPunkt, GrafPunkt, double, double*) → *Unterpunkt*
  überschreibt die Methode der Klasse *GrafZustand*.
  Fügt eine neue Partition hinzu wie addSubPoint, allerdings ist der angegebene Offset in Koordinaten des angegebenen Unterzustands. Ist der übergebene Zustand kein direkter Unterzustand dieses Zustands, wird `Null` zurückgegeben.
+ removeSubPoint (*GrafPunkt*) → *void*
  überschreibt die Methode der Klasse *GrafZustand*.
  Entfernt eine Partition. Im logischen Modell wird dieser Vorgang wiederholt.
+ getObjectAt (*double, double, int, double*) → *GrafTyp*
  überschreibt die Methode der Klasse *GrafZustand*.
  Gibt den grafischen Punkt zurück, der sich an den angegebenen Koordinaten (relativ zur linken oberen Ecke dieses Zustands) befindet. Bei der Suche wird bis in die angegebene Tiefe weitergegangen. Dieser Zustand selbst wird nicht zurückgegeben, sondern nur seine Partitionen. Eine Tiefe von `0` heißt, dass nur die Partitionen dieses Zustands betrachtet werden sollen, aber nicht die Partitioner oder Pfeile. Gibt `Null` zurück, wenn kein entsprechendes Objekt gefunden wird.
+ probeDepth () → *int*
  überschreibt die Methode der Klasse *GrafZustand*.
  Gibt die maximale Tiefe unterhalb dieses Zustands an.
+ draw (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*
  überschreibt die Methode der Klasse *GrafZustand*.
  Zeichnet diesen grafischen Punkt und Unterzustände bis zur angegebenen Tiefe. Eine Tiefe von `0` heißt dabei, dass nur dieser Zustand - ohne Partitionen - gezeichnet werden soll, bei `1` werden die Partitionen und deren direkte Unterzustände gezeichnet. Der angegebene Offset ist der Punkt, an dem die linke obere Ecke dieses Zustands effektiv gezeichnet werden soll.
+ hasSubPoints () → *boolean*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt an, ob dieser Zustand Unterpunkte besitzt. Dies ist bei einem AND-Zustand immer der Fall.

+ isValidPartitioner (*Partitioner*) → *boolean*

  Gibt an, ob Anfangs- und Endpunkt des übergebenen Partitioners innerhalb dieses Zustands liegen.

+ addPartitioner (*Partitioner*) → *void*

  Fügt einen Partitioner hinzu. Es wird angenommen, dass dieser Partitioner ein für diesen Zustand gültiger ist.

+ isValidHMove (*double, double*) → *boolean*

  überschreibt die Methode der Klasse *GrafZustand*.
  Gibt an, ob die übergebene horizontale Bewegung eines Partitioners gültig ist.

+ isValidVMove (*double, double*) → *boolean*

  überschreibt die Methode der Klasse *GrafZustand*.
  Gibt an, ob die übergebene vertikale Bewegung eines Partitioners gültig ist.

+ doHMove (*double, double, double, double*) → *void*

  überschreibt die Methode der Klasse *GrafZustand*.
  Führt eine horizontale Bewegung eines Partitioners durch.

+ doVMove (*double, double, double, double*) → *void*

  überschreibt die Methode der Klasse *GrafZustand*.
  Führt eine vertikale Bewegung eines Partitioners durch.

+ copyToNewSuper (*GrafZustand, Point2D.Double, Hashtable[]*) → *GrafZustand*

  überschreibt die Methode der Klasse *GrafZustand*.
  Kopiert diesen AND-Zustand in den neuen Oberzustand. Dabei werden auch Partitioner kopiert.

+ delete () → *void*

  überschreibt die Methode der Klasse *GrafZustand*.
  Löscht diesen Zustand.

+ revertToOR () → *GrafZustand*

  Reduziert diesen AND-Zustand zu einem OR-Zustand. Dies geschieht dadurch, dass die Partitionen aufgelöst werden und die in ihnen enthaltenen Zustände direkt in den (ehemaligen) AND-Zustand eingefügt werden.

+ save (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *GrafZustand*.
  Speichert diesen AND-Zustand im EasyFileWriter.

D.9.5 Klasse GrafHook

extends *score.GrafSCM.GrafTyp*
Diese Klasse ist ein Knotenpunkt eines Grafischen Pfeils.

Attribute

+$ START:*int* CONST – Konstante für Starthooks

+$ END:*int* CONST – Konstante für Endhooks

+$ PLAIN_MID:*int* CONST – Konstante für normale mittlere Hooks

+$ CONN_IN:*int* CONST – Konstante für mittlere Hooks, die mehrere eingehende Verbindungen haben

+$ CONN_OUT:*int* CONST – Konstante für mittlere Hooks, die mehrere ausgehende Verbindungen haben

+$ DELSCOPE_HOOK:*int* CONST – Konstante, die angibt, dass nur dieser Hook gelöscht werden soll

+$ DELSCOPE_SECTION:*int* CONST – Konstante, die angibt, dass der Abschnitt dieses Hooks gelöscht werden soll

+$ DELSCOPE_ARROW:*int* CONST – Konstante, die angibt, dass der Pfeil dieses Hooks gelöscht werden soll

Konstruktoren

+ GrafHook (*GrafPfeil, GrafPunkt, int, Point2D.Double*)

  Erschafft einen neuen Hook zum übergebenen Pfeil. Er hat den übergebenen HookTyp und bezüglich des übergebenen Punktes den angegebenen Offset.

+ GrafHook (*GrafPfeil, GrafPunkt, int, Point2D.Double, Integer*)

  Erschafft einen neuen Hook, dessen Pfeil den Text mit dem angegebenen Identifier als Label zugewiesen bekommt.

Methoden

+ setHookType (*int*) → *void*

  Setzt den HookTyp gemäß den Konstanten dieser Klasse.

+ getArrow () → *GrafPfeil*

  Gibt den Pfeil zurück, zu dem dieser Hook gehört.

+ getOffset () → *Point2D.Double*

  Gibt den Offset dieses Hooks bezüglich seines Oberzustands zurück.

+ setTo (*GrafPunkt, Point2D.Double*) → *void*

  Bewegt diesen Hook zum angegebenen Oberzustand mit dem übergebenen Offset. Es findet keinerlei Prüfung statt.

+ moveTo (*double, double, GrafPunkt*) → *void*

  Bewegt den Hook an die angegebene Stelle bezüglich des übergebenen Punktes. Der Hook wird in den entsprechenden Zustand einsortiert, wobei die SuperList des Pfeils benutzt aber nicht verändert wird.

+ moveToNewAnchor (*Point2D.Double, GrafPunkt*) → *void*

  Bewegt diesen Hook an die übergebene Stelle des neuen Ankers.

+ relocalize () → *void*

  Stellt sicher, dass sich der Hook im richtigen Zustand befindet, ohne ihn zu bewegen.

+ getAnchor () → *GrafPunkt*

  Gibt den Oberzustand dieses Hooks zurück.

+ getAnchorAtPos () → *GrafPunkt*

  Gibt den Anker (Point or Partition) zurück, der sich an der Position dieses Hooks befindet.

+ isStart () → *boolean*

  Gibt zurück, ob dieser Hook ein Startpunkt ist.

+ isPlainMid () → *boolean*

Gibt zurück, ob dieser Hook ein normaler mittlerer Hook ist.

+ isEnd () → *boolean*
  Gibt zurück, ob dieser Hook ein Endpunkt ist.
+ isConnectIn () → *boolean*
  Gibt zurück, ob dieser Hook ein mittlerer Hook ist, der mehrere eingehende Verbindungen hat.
+ isConnectOut () → *boolean*
  Gibt zurück, ob dieser Hook ein mittlerer Hook ist, der mehrere ausgehende Verbindungen hat.
+ isConnect () → *boolean*
  Gibt zurück, ob dieser Hook ein erweiterter mittlerer Hook ist.
+ isMid () → *boolean*
  Gibt zurück, ob dieser Hook ein mittlerer Hook ist; entweder normal oder erweitert.
+ mayBeConnectIn () → *boolean*
  Gibt zurück, ob dieser Hook mehrere eingehende Verbindungen haben darf.
+ mayBeConnectOut () → *boolean*
  Gibt zurück, ob dieser Hook mehrere ausgehende Verbindungen haben darf.
+ maybeArrowSource () → *boolean*
  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt zurück, ob dieser Hook mehrere ausgehende Verbindungen haben darf.
+ maybeArrowTarget () → *boolean*
  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt zurück, ob dieser Hook mehrere eingehende Verbindungen haben darf.
+ getObjectAt (*double, double, double*) → *GrafTyp*
  Gibt das Objekt an den übergebenen Koordinaten (des Oberzustands) zurück; d. h. diesen Hook, wenn er sich nah genug daran befindet.
+ makeDouble () → *void*
  Fügt einen neuen Knoten neben diesem Knoten ein. Bei Mittelpunkten wird dieser Knoten durch zwei benachbarte ersetzt.
+ addLabel () → *void*
  Fügt zu diesem Hook den Text hinzu, der das Label des zugehörigen Pfeils ist.
+ removeText (*GrafText*) → *void*
  überschreibt die Methode der Klasse *GrafTyp*.
  Entfernt den angegebenen Text von diesem Hook; der zugehörige Pfeil hat dann auch kein Label mehr.
+ getPosOfNeighbours (*GrafModellSicht*) → *Vector*
  Gibt die Zeichenkoordinaten aller benachbarten Hooks zurück.
+ getDrawCoords (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten dieses Hooks zurück. Kann `Null` sein, wenn sich dieser Hook nicht innerhalb des Oberzustands der übergebenen Sicht befindet.
+ getOuterDrawCoords (*GrafModellSicht*) → *Point2D.Double*
  Gibt die Koordinaten dieses Hooks zurück, bezüglich des angegebenen Views, in dem dieser Hook eigentlich nicht mehr sichtbar ist, da er nicht unterhalb des Oberzustands liegt.
+ delete () → *void*
  überschreibt die Methode der Klasse *GrafTyp*.
  Löscht diesen Hook. Wenn es sich um einen Start- oder Endhook handelt, wird auch der entsprechende Abschnitt gelöscht; wenn es sich um einen Connecthook handelt, wird der gesamte Pfeil gelöscht.
+ deleteSection () → *void*
  Löscht den Abschnitt, zu dem dieser Hook gehört.
+ deleteArrow () → *void*

Löscht den Pfeil, zu dem dieser Hook gehört.

+ delScope () → *int*
  Gibt zurück, welche Objekte von diesem Hook aus gelöscht werden können.
+ mayDeleteHook () → *boolean*
  Gibt zurück, ob dieser Hook aus seinem Abschnitt herausgelöscht werden darf, ohne dass dadurch der gesamte Abschnitt gelöscht werden muss.
+ mayDeleteSection () → *boolean*
  Gibt zurück, ob der Abschnitt, zu dem dieser Hook gehört, gelöscht werden darf, ohne dass dadurch der gesamte Pfeil gelöscht werden muss.
+ translate (*Hashtable[], GrafPfeil*) → *GrafHook*
  Gibt eine Übersetzung dieses Hooks zurück, die zum übergebenen Pfeil gehört.
+ save (*EasyFileWriter*) → *void*
  Speichert diesen Hook im EasyFileWriter.

D.9.6 Klasse GrafKonnektor

extends *score.GrafSCM.GrafPunkt*
Diese Klasse stellt einen Konnektor dar. Alle drei Typen von Konnektor sind hier zusammengefasst.

Attribute

+$ UNIVERSAL:*int* CONST – Konstante für Universal-Konnektoren
+$ HISTORY:*int* CONST – Kontante für History-Konnektoren, die keine Deep-History ausführen
+$ DEEP_HISTORY:*int* CONST – Konstante für Deep-History-Konnektoren
+$ DEFENTRY:*int* CONST – Konstante für Default-Einstieg-Konnektoren

Konstruktoren

+ GrafKonnektor (*Dimension.Double, int*)
  Erschafft einen neuen Konnektor mit angegebener Größe und Typ.
+ GrafKonnektor (*Dimension.Double, Konnektor*)
  Erschafft einen neuen Konnektor aus dem übergebenen logischen Konnektor.

Methoden

+ setSize (*Dimension.Double*) → *void*
  überschreibt die Methode der Klasse *GrafPunkt*.
  Setzt die Größe des Konnektors. Es wird davon ausgegangen, dass diese Größe gültig ist.
+ putOntoRim (*Point2D.Double*) → *Point2D.Double*
  Legt den angegebenen Punkt auf den Rand dieses Konnektors.
+ draw (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*
  überschreibt die Methode der Klasse *GrafPunkt*.
  Zeichnet den Konnektor, der angegebene Offset gibt die Position der linken oberen Ecke direkt an.
+ isSECorner (*double, double, int, double*) → *int*
  überschreibt die Methode der Klasse *GrafPunkt*.
  Gibt 1 zurück, wenn der Punkt in der Nähe der rechten unteren Ecke liegt, -1 wenn das nicht der Fall ist, und 0 wenn dieser Zustand nicht betroffen ist.
+ getPosAtBorder (*Point2D.Double*) → *Point2D.Double*
  überschreibt die Methode der Klasse *GrafPunkt*.
  Legt den angegebenen Punkt auf den Rand dieses Konnektors.
+ maybeArrowSource () → *boolean*

überschreibt die Methode der Klasse *GrafTyp*.
Gibt zurück, ob dieser Konnektor ein Startpunkt für einen Pfeil sein darf; dies ist immer wahr.

\+ maybeArrowTarget () → *boolean*
überschreibt die Methode der Klasse *GrafTyp*.
Gibt zurück, ob dieser Konnektor ein Endpunkt für einen Pfeil sein darf.

\+ copyToNewSuper (*GrafZustand, Hashtable[]*) → *GrafZustand*
überschreibt die Methode der Klasse *GrafPunkt*.
Kopiert diesen Konnektor in den übergebenen neuen Oberzustand. Der Rückgabewert ist immer `Null`.

\+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *GrafPunkt*.
Speichert diesen Konnektor im EasyFileWriter.

D.9.7 Klasse GrafModellSicht

extends *score.util.Observable*
implements *score.util.Observer*

innere Klassen: *GrafModellSicht.visHook, GrafModellSicht.visText*
Diese Klasse speichert die Eigenschaften, die ein Zustand hat, wenn er als oberster Zustand einer Ansicht angezeigt wird.

Attribute

+$ undo:*UndoManager* – der UndoManager, dem alle Sichten Änderungen mitteilen

+$ doRefresh:*boolean* – Wenn diese Variable auf `false` gesetzt ist, wird kein Refresh an die Observer dieser Sicht weitergegeben.

+$ DEF_SIZE:*Dimension.Double* `CONST` – Standardgröße einer Sicht

+$ BORDER:*Dimension* `CONST` – Standardgröße des Randes einer Sicht

Konstruktoren

\+ GrafModellSicht (*GrafZustand*)
Erschafft eine Sicht auf den übergebenen Zustand. Die Sicht wird mit neuen default-Properties erschaffen.

Methoden

\+ replaceObservable (*Observable, Observable*) → *boolean*
Implementation des Observer-Interfaces

\+ setErrorPrinter (*ErrorPrinter*) → *void*
Setzt den ErrorPrinter dieser Sicht.

+$ setUndoManager (*UndoManager*) → *void*
Setzt den UndoManager für alle Sichten.

+$ setModell (*GrafSCModell*) → *void*
Setzt das Modell, auf das alle Sichten zugreifen werden.

+$ getModell () → *GrafSCModell*
Gibt das grafische Statechart Modell zurück.

+$ setMarkedObject (*GrafTyp*) → *void*
Setzt den übergebenen GrafTyp (im grafischen Modell) als aktives Objekt.

+$ getMarkedObject () → *GrafTyp*
Gibt das momentan aktive Objekt aus dem Grafischen Modell zurück.

\+ getTopState () → *GrafZustand*

Gibt den Oberzustand dieser Sicht zurück.

+ setTopState (*GrafZustand*) → *void*
  Setzt den Oberzustand dieser Sicht.
+ isTopLevelView () → *boolean*
  Gibt an, ob diese Sicht sich auf den virtuellen Oberzustand bezieht.
+ setDepth (*int*) → *void*
  Setzt die Zeichentiefe.
+ getDepth () → *int*
  Gibt die Zeichentiefe zurück.
+ setZoom (*double*) → *void*
  Setzt die Vergrößerung
+ getZoom () → *double*
  Gibt die Vergrößerung zurück
+ setOffset (*Point*) → *void*
  Setzt den Zeichenoffset des Oberzustands dieser Sicht.
+ getOffset () → *Point2D.Double*
  Gibt den Zeichenoffset des Oberzustands dieser Sicht zurück.
+ showLabels () → *boolean*
  Gibt an, ob Texte gezeichnet werden.
+ showLabels (*boolean*) → *boolean*
  Setzt die Angabe, ob Texte gezeichnet werden sollen.
+ showIDs () → *boolean*
  Gibt an, ob Identifier gezeichnet werden.
+ showIDs (*boolean*) → *boolean*
  Setzt die Angabe, ob Identifier gezeichnet werden sollen.
+ showGrid (*boolean*) → *boolean*
  Schaltet das Gitter für diese Sicht ein oder aus.
+ showGrid () → *boolean*
  Gibt zurück, ob diese Sicht gerade ein Gitter anzeigt.
+ showHooks (*boolean*) → *boolean*
  Schaltet die Hooks für diese Sicht ein oder aus.
+ showHooks () → *boolean*
  Gibt zurück, ob diese Sicht gerade Hooks anzeigt.
+ getViewDimension () → *Dimension.Double*
  Gibt die Größe des Oberzustandes zurück oder `Null`, wenn diese Sicht sich auf den virtuellen Oberzustand bezieht.
+ getBoundingBox () → *Dimension.Double*
  Gibt für den virtuellen Oberzustand die Größe einer umgebenden Box der Unterzustände zurück, oder `Null`, wenn es noch keine solchen Zustände gibt. Für andere Sichten wird die Größe des Oberzustand zurückgegeben.
+ SEOfView () → *Dimension.Double*
  Gibt die Zeichenposition der rechten unteren Ecke zurück.
+ insertNewState (*Point2D.Double, Point2D.Double*) → *int*
  Erzeugt einen neuen Zustand, der durch die übergebenen Punkte beschrieben wird. Diese sind in Zeichenkoordinaten gegeben. Wenn diese Punkte keine gültige Kombination darstellen, wird eine negative Zahl zurückgegeben.
+ insertNewConn (*Point2D.Double, Point2D.Double, int*) → *int*

Fügt einen neuen Konnektor vom übergebenen Typ ein.

+ insertNew1_1Arrow (*GrafPunkt, GrafPunkt, Point2D.Double, Point2D.Double*) → *void*
  Fügt einen neuen 1-1 Pfeil zwischen den übergebenen Grafischen Punkten ein.
+ insertNewReflectiveArrow (*GrafPunkt*) → *void*
  Fügt zum übergebenen Punkt einen neuen rückbezüglichen Pfeil hinzu.
+ insertNewText (*Point2D.Double, String*) → *int*
  Fügt einen neuen Text an der angegebenen Position hinzu. Hiermit werden Texte nur in Zuständen eingefügt, nicht aber Texte zu Hooks.
+ connectIn (*GrafHook, GrafPunkt, Point2D.Double*) → *void*
  Fügt zum übergebenen Hook eine neue eingehende Verbindung hinzu.
+ connectOut (*GrafHook, GrafPunkt, Point2D.Double*) → *void*
  Fügt zum übergebenen Hook eine neue ausgehende Verbindung hinzu.
+ getDepthToTop (*GrafTyp*) → *int*
  Gibt die Tiefe des übergebenen Objektes zum Oberzustand zurück.
– isValidMove (*GrafPunkt, Point2D.Double*) → *boolean*
  Gibt zurück, ob der angegebene Zustand an die angegebene Stelle bewegt werden darf.
– isValidMove (*Partitioner, Point2D.Double*) → *boolean*
  Gibt zurück, ob der angegebene Partitioner an die angegebene Stelle bewegt werden darf.
– isValidMove (*GrafHook, Point2D.Double*) → *boolean*
  Gibt zurück, ob der angegebene Hook an die angegebene Stelle bewegt werden darf.
+ isValidSize (*GrafPunkt, Dimension.Double*) → *boolean*
  Gibt zurück, ob die angegebene Größe für den übergebenen GrafPunkt gültig ist.
+ isValidSize (*GrafPunkt, Dimension*) → *boolean*
  Gibt zurück, ob die angegebene Größe für den angegebenen Zustand gültig ist.
+ isValidMove (*GrafTyp, Point*) → *boolean*
  Gibt zurück, ob die angegebene Bewegung für das übergebene Objekt gültig ist.
– moveTo (*GrafPunkt, Point2D.Double*) → *int*
  Bewegt den angegebenen Zustand an die in Zeichenkoordinaten angegebene Stelle. Es wird davon ausgegangen, dass diese Bewegung gültig ist. Die neue Tiefe des Zustandes wird zurückgegeben.
– moveTo (*GrafHook, Point2D.Double*) → *int*
  Bewegt den Hook an die übergebene Position.
– moveTo (*Partitioner, Point2D.Double*) → *int*
  Bewegt den Partitioner an die übergebene Position.
– moveTo (*GrafText, Point2D.Double*) → *int*
  Bewegt den Text an die übergebene Position.
+ moveTo (*GrafTyp, Point*) → *int*
  Bewegt das übergebene Objekt an die angegebene Position. Es wird davon ausgegangen, dass diese Bewegung gültig ist.
+ isValidCopy (*GrafZustand, Point*) → *boolean*
  Gibt an, ob die übergebene Position für eine Kopie gültig ist.
+ copyStateTo (*GrafZustand, Point, TranslationWindow*) → *void*
  Kopiert den angegebenen Zustand an die übergebene Stelle. Der Benutzer wird mit dem übergebenen TranslationWindow gebeten, Ereignisse und Variablen umzubenennen. Die Kontrolle geht dann an dieses Fenster, das die Methode update entsprechend aufruft, wenn es beendet wird.
+ delete (*GrafTyp*) → *void*
  Löscht den übergebenen GrafTyp.
+ deleteArrowOfHook (*GrafHook*) → *void*

Löscht den Pfeil, der zum übergebenen Hook gehört.

+ deleteSectionOfHook (*GrafHook*) → *void*
  Löscht den Abschnitt, zu dem der übergebene Hook gehört.
+ doubleHook (*GrafHook*) → *void*
  vergleiche: makeDouble() in Klasse *score.GrafSCM.GrafHook*
  Verdoppelt den angegebenen Hook.
+ labelArrow (*GrafHook*) → *void*
  Fügt zum angegebenen Hook ein Pfeil-Label hinzu.
+ identifyPoint (*Point, int*) → *GrafTyp*
  Gibt den GrafTyp zurück, der sich an angegeben Position befindet; diese Methode erwartet unskalierte Mauskoordinaten.
+ identifyPoint (*Point2D.Double, double*) → *GrafTyp*
  Gibt den GrafTyp des Objektes zurück, das sich an den angegebenen Koordinaten befindet, die Zeichenkoordinaten sind. Es wird bis in die eingestellte Tiefe vorgedrungen, wobei eine Tiefe von `0` heißt, dass nur der Zustand selbst betrachtet wird. Wenn sich der Punkt nicht innerhalb des Oberzustandes befindet, wird `Null` zurückgegeben.
+ getHookAt (*Point2D.Double, double*) → *GrafHook*
  Gibt einen Hook zurück, falls sich an der angegebenen Stelle einer befindet, sonst `Null`.
+ drawsHookAt (*GrafHook, Point2D.Double*) → *void*
  Sagt der Sicht, dass der übergebene Hook an der angegebenen Stelle gezeichnet wird.
+ getTextAt (*Point2D.Double, double*) → *GrafText*
  Gibt einen Text zurück, falls sich an der angegebenen Stelle einer befindet, sonst `Null`.
+ drawsTextAt (*GrafText, Point2D.Double*) → *void*
  Sagt der Sicht, dass der übergebene Text an der angegebenen Stelle gezeichnet wird.
+ getPointAt (*Point, int*) → *GrafPunkt*
  Gibt den GrafPunkt an der angegebenen Position zurück; diese Methode erwartet unskalierte Mauskoordinaten.
+ getPointAt (*Point*) → *GrafPunkt*
  Gibt den GrafPunkt an der angegebenen Position zurück; diese Methode erwartet unskalierte Mauskoordinaten.
+ getPointAt (*Point2D.Double, double*) → *GrafPunkt*
  Gibt den Zustand zurück, der sich an den angegebenen Zeichenkoordinaten befindet. Ist das Objekt kein Zustand (oder wird kein Objekt gefunden), wird `Null` zurückgegeben.
+ getPointAt (*Point2D.Double*) → *GrafPunkt*
  Gibt den grafischen Punkt zurück, der sich an den angegebenen Zeichenkoordinaten befindet. Wird eine Partition gefunden, so wird der zugehörige AND-Zustand zurückgegeben.
+ getPointOrPartitionAt (*Point2D.Double*) → *GrafPunkt*
  Gibt den grafischen Punkt zurück, der sich an den angegebenen Zeichenkoordinaten befindet. Hierbei werden auch Partitionen beachtet.
+ getKoordsOfSub (*Point2D.Double, GrafPunkt*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten eines Punktes zurück, der in lokalen Koordinaten von point angegeben ist.
+ getLocalOffset (*Point*) → *Point2D.Double*
  Gibt den Offset zurück, den der übergebene (unskalierte) Punkt im untersten sichtbaren Zustand hat.
+ getLocalOffset (*Point2D.Double*) → *Point2D.Double*
  Gibt den Offset zurück, den der übergebene Punkt im untersten sichtbaren Zustand hat.
+ localizeToSub (*Point2D.Double, GrafPunkt*) → *Point2D.Double*

Gibt den Offset zurück, den der übergebene Punkt im übergebenen Zustand hat.

+ getLocalAlignment (*Point2D.Double*) → *int*

  Gibt die Ausrichtung zurück, den der übergebene Punkt im untersten sichtbaren Zustand hat.

+ isAtBorder (*Point2D.Double*) → *boolean*

  Gibt zurück, ob sich der angegebene Punkt im untersten sichtbaren Zustand in der Nähe des Randes befindet. Was in diesem Sinne der Begriff Nähe bedeutet, wird im GrafPunkt dadurch geregelt, ab wann getLocalAlignment nicht mehr javax.swing.SwingConstants.CENTER zurückgibt.

+ getPosAtBorderOf (*Point, GrafPunkt*) → *Point2D.Double*

  Gibt eine Position auf dem Rand des übergebenen grafischen Punktes zurück, die für den übergebenen (unskalierten) Punkt nahe liegt.

+ getPosAtBorderOf (*Point2D.Double, GrafPunkt*) → *Point2D.Double*

  Gibt eine Position auf dem Rand des übergebenen grafischen Punktes zurück, die für den übergebenen Punkt nahe liegt.

+ isValidConnPos (*Point2D.Double, Point2D.Double*) → *boolean*

  Gibt zurück, ob das übergebene Rechteck eine gültige Position für einen Konnektor ist.

– getSuperOf (*Point2D.Double, Point2D.Double*) → *GrafPunkt*

  Gibt den grafischen Punkt zurück, der oberhalb des angegebenen Rechtecks liegt.

+ isSECorner (*Point, int*) → *boolean*

  Gibt zurück, ob dieser (unskalierte) Punkt im entsprechenden Zustand in der Nähe der rechten unteren Ecke liegt.

+ isSECorner (*Point2D.Double, double*) → *boolean*

  Gibt zurück, ob dieser Punkt im entsprechenden Zustand in der Nähe der rechten unteren Ecke liegt.

+ getPosition (*GrafPunkt*) → *Point2D.Double*

  Gibt die Position zurück, die der übergebene GrafPunkt bezüglich des Oberzustands dieser Sicht hat.

+ draw (*Painter*) → *void*

  Zeichnet den Hauptzustand mit der eingestellten Tiefe in den übergebenen Painter.

+ hasBeenDrawn (*GrafPfeil*) → *boolean*

  Gibt an, ob der übergebene Pfeil bereits gezeichnet wurde.

+ drawsArrow (*GrafPfeil*) → *void*

  Sagt der Sicht, dass der übergebene Pfeil gezeichnet wurde.

+ getPartitioners (*Point2D.Double*) → *Partitioner*

  Gibt die möglichen Partitioner bei einem gegebenen Punkt zurück. Liegt der Punkt nicht im Hauptzustand, wird `Null` zurückgegeben.

+ getHorizPartitioner (*Point*) → *Partitioner*

  Gibt einen möglichen horizontalen Partitioner oder `Null` zurück.

+ getVertPartitioner (*Point*) → *Partitioner*

  Gibt einen möglichen vertikalen Partitioner oder `Null` zurück.

+ getPosOfNeighbours (*GrafHook*) → *Point2D.Double*

  Gibt einen Array der Zeichenpositionen der Hooks neben dem übergebenen Hook zurück.

+ update (*Observable, Object*) → *void*

  Implementation des Observer-Interfaces. Führt ein Refresh durch, solange sie nicht von einem TranslationWindow aufgerufen wird. In diesem Fall ist das übergebene Argument der GrafZustand, der kopiert wurde.

+ refresh (*Observable*) → *void*

  Erneuert die Darstellung aller Observer dieser Sicht.

+ refresh () → *void*

Erneuert die Darstellung dieser Sicht.

+ refreshModell () → *void*
  Ruft die Methode refresh des Modells auf.
+ save (*EasyFileWriter*) → *void*
  Speichert diese Sicht zu ihrem Zustand im EasyFileWriter.

D.9.8 Klasse GrafModellSicht.visHook

extends *java.lang.Object*
Diese interne Klasse steht für einen Hook, der gezeichnet wurde. Unabhängig von seiner Lage in der Hierarchie kann damit einfach getestet werden, ob die Maus gerade auf ihn zeigt.

Attribute

+ point:*Point2D.Double* – die Zeichenkoordinaten des Hook
+ hook:*GrafHook* – der gezeichnete Hook

Konstruktoren

+ GrafModellSicht.visHook (*GrafModellSicht, Point2D.Double, GrafHook*)
  Erschafft ein neues Paar von Hook und Zeichenpunkt.

D.9.9 Klasse GrafModellSicht.visText

extends *java.lang.Object*
Diese interne Klasse steht für einen Text, der gezeichnet wurde. Unabhängig von seiner Lage in der Hierarchie kann damit einfach getestet werden, ob die Maus gerade auf ihn zeigt.

Attribute

+ point:*Point2D.Double* – die Zeichenkoordinaten des Textes
+ text:*GrafText* – der gezeichnete Text

Konstruktoren

+ GrafModellSicht.visText (*GrafModellSicht, Point2D.Double, GrafText*)
  Erschafft ein neues Paar von Text und Zeichenpunkt.

D.9.10 Klasse GrafPfeil

extends *score.GrafSCM.GrafTyp*
implements *score.util.Observer*

Diese Klasse beschreibt einen grafischen Pfeil.

Konstruktoren

+ GrafPfeil (*GrafPunkt, GrafPunkt, Point2D.Double, Point2D.Double*)
  Erzeugt einen neuen 1-1 Pfeil mit gegebenem grafischen Start- und Endpunkt.

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*
  Implementation des Observer-Interfaces
+ addStartSect (*GPAbschnitt*) → *void*
  Fügt die übergebene Section in diesen Pfeil ein. Der Anfangspunkt des logischen Pfeils wird entsprechend gesetzt. Dieser Pfeil wird auch im grafischen und logischen Punkt hinzugefügt. Der Konnektor-Hook muss schon gesetzt sein.
+ addEndSect (*GPAbschnitt*) → *void*

Fügt den übergebenen Abschnitt in diesen Pfeil ein. Der Endpunkt des logischen Pfeils wird entsprechend gesetzt. Dieser Pfeil wird auch im grafischen und logischen Punkt hinzugefügt. Der Konnektor-Hook muss schon gesetzt sein.

\+ setConnect (*GrafHook*) → *void*

Setzt den KonnektorHook für diesen GrafPfeil.

\+$ getGrafPfeil (*int*) → *GrafPfeil*

Gibt den grafischen Pfeil mit dem übergebenen Identifier zurück.

\+$ getAll () → *Collection*

Gibt eine Collection mit allen grafischen Pfeilen zurück.

\+ setLogic (*LogPfeil*) → *void*

Setzt das logische Gegenstück dieses grafischen Pfeils.

\+ update (*Observable, Object*) → *void*

Implementation des Observer-Interfaces; erneuert ggf. auch ein vorhandenes Label.

\+ getLogic () → *LogPfeil*

Gibt das logische Gegenstück dieses grafischen Pfeils zurück.

\+ hasConnect () → *boolean*

Gibt zurück, ob dieser Pfeil einen Konnektor Hook hat.

\+ getConnect () → *GrafHook*

Gibt den Konnektor Hook zurück, oder `Null`, wenn keiner existiert.

\+ translate (*Hashtable[]*) → *GrafPfeil*

Gibt eine übersetzung dieses Pfeils zurück.

\+ toString () → *String*

überschreibt die Methode der Klasse *MarkableTyp*.
Gibt eine String-Darstellung dieses grafischen Pfeils zurück.

\+ getLabelText () → *String*

Gibt den Text zurück, den ein Label für diesen Pfeil anzeigen sollte.

\+ setLabel (*GrafText*) → *void*

Setzt den übergebenen Text als Label für diesen Pfeil. Ein vorher eventuell vorhandenes Label wird dabei gelöscht.

\+ connectIn (*GrafHook, GrafPunkt, Point2D.Double*) → *void*

Macht den übergebenen Hook zu einem In-Connector, falls er noch keiner ist. In jedem Fall wird ein neuer Abschnitt vom übergebenen Startpunkt hinzugefügt.

\+ connectOut (*GrafHook, GrafPunkt, Point2D.Double*) → *void*

Macht den übergebenen Hook zu einem Out-Connector, falls er noch keiner ist. In jedem Fall wird ein neuer Abschnitt zum übergebenen Endpunkt hinzugefügt.

\+ draw (*Painter, GrafModellSicht*) → *void*

Zeichnet diesen Pfeil in den übergebenen Painter für die übergebene GrafModellSicht.

\+ getSectWith (*GrafHook*) → *GPAbschnitt*

Gibt den Abschnitt zurück, in dem dieser Hook liegt, oder `Null`, wenn kein solcher gefunden wurde. Es wird lediglich der erste Treffer zurückgegeben und nicht getestet, ob mehrere existieren.

\+ getPosOfConnIn (*GrafModellSicht*) → *Vector*

Gibt die Zeichenkoordinaten der Nachbarzuständes eines In-Connect-Hook zurück. Es wird davon ausgegangen, dass dieser Pfeil einen solchen besitzt.

\+ getPosOfConnOut (*GrafModellSicht*) → *Vector*

Gibt die Zeichenkoordinaten der Nachbarzuständes eines Out-Connect-Hook zurück. Es wird davon ausgegangen, dass dieser Pfeil einen solchen besitzt.

\+ getStartStates () → *Vector*

Gibt einen Vector mit den Startzuständen dieses Pfeils zurück. Dieser ist weder `Null` noch leer.

+ getEndStates () → *Vector*
  Gibt einen Vector mit den Endzuständen dieses Pfeil zurück. Dieser ist weder `Null` noch leer.
+ hasHookOutside (*GrafPunkt*) → *boolean*
  Gibt an, ob einer der Abschnitte einen Hook besitzt, der außerhalb des übergebenen Punktes liegt.
+ hasStartOutside (*GrafPunkt*) → *boolean*
  Gibt an, ob einer der Startzustände außerhalb des übergebenen Punktes liegt.
+ hasEndOutside (*GrafPunkt*) → *boolean*
  Gibt an, ob einer der Endzustände außerhalb des übergebenen Punktes liegt.
+ getSectStartingAt (*GrafPunkt*) → *GPAbschnitt*
  Gibt die Abschnitte zurück, die bei dem übergebenen Punkt beginnen, oder `Null`, wenn es keine solchen gibt.
+ getSectEndingAt (*GrafPunkt*) → *GPAbschnitt*
  Gibt die Abschnitte zurück, die bei dem übergebenen Punkt enden, oder `Null`, wenn es keine solchen gibt.
+ getCoordsOfSecondTo (*GrafModellSicht, GrafPunkt*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des zweiten Punktes nach dem übergebenen Zustand zurück, oder `Null`, wenn dieser Punkt nicht als Startzustand gefunden wird.
+ getCoordsOfPenultimateTo (*GrafModellSicht, GrafPunkt*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des vorletzten Punktes vor dem übergebenen Punkt zurück, oder `Null`, wenn dieser Punkt nicht als Endpunkt gefunden wird.
+ getCoordsOfFirstTouching (*GrafModellSicht, GrafPunkt*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des Anfangshooks dieses Pfeiles auf dem übergebenen Punkt zurück.
+ getCoordsOfLastTouching (*GrafModellSicht, GrafPunkt*) → *Point2D.Double*
  Gibt die Zeichenkoordinaten des Endhooks dieses Pfeiles auf dem übergebenen Punkt zurück.
+ addPosOfCrossing (*Vector, GrafPunkt, Vector, GrafModellSicht*) → *void*
  Fügt zu dem Vector die Positionen der Hooks hinzu, deren Verbindung den Rand des übergebenen Zustands schneidet.
+ addHooksJustOutside (*Vector, GrafPunkt*) → *void*
  Fügt zum übergebenen Vector die Hooks dieses Pfeils hinzu, die gerade außerhalb des übergebenen GrafPunkts liegen.
+ getMidVisibleContainers () → *Vector*
  Gibt einen Vector mit den Zuständen zurück, in denen sämtliche Hooks dieses Pfeils liegen. Dabei wird nur die Lage des Hooks, und nicht der tatsächliche Anker betrachtet.
+ getAllSuperstates () → *Vector*
  Gibt einen Vector mit allen Zuständen zurück, die oberhalb mindestens eines der Start- oder Endpunkte dieses Pfeils liegen.
+ getHooksOf (*GrafPunkt*) → *Vector*
  Gibt die Hooks dieses Pfeils zurück, die am übergebenen GrafPunkt liegen.
+ getSuperList () → *SuperList*
  Gibt eine SuperList zurück, die alle Start- und Endzustände dieses Pfeils schon beachtet.
+ getHookedSuperList () → *SuperList*
  Gibt eine SuperList für diesen Pfeil zurück, die auch die Lage sämtlicher Hooks beachtet.
+ eraseSuperList () → *void*
  Löscht die SuperList dieses Pfeils.
+ relocalize () → *void*

Stellt sicher, dass sich alle Hooks dieses Pfeils in den richtigen Oberzuständen befinden.

+ redirect (*GrafPunkt, GrafPunkt*) → *void*

Lenkt diesen Pfeil von einem Punkt auf einen anderen um. Es wird davon ausgegangen, dass sich die beiden Punkte in ihrer Lage entsprechen. Dabei werden die Hooks und die Logik mitgeführt. Es wird jedoch nur eine Section beachtet.

+ delete () → *void*

überschreibt die Methode der Klasse *GrafTyp*.
Löscht diesen Pfeil. Er wird auch aus allen grafischen Start- und Endzuständen entfernt.

+ removeSect (*GPAbschnitt*) → *void*

Entfernt den übergebenen Abschnitt. Wenn es sich um einen 1-1 Pfeil, den Startabschnitt eines 1-N Pfeils oder den Endabschnitt eines N-1 Pfeils handelt, wird dabei der gesamte Pfeil gelöscht. Ansonsten wird ggf. das logische Gegenstück auch auf einen 1-1 Pfeil zurückgesetzt.

– revertTo11 () → *void*

Reduziert das logische Gegenstück auf einen 1-1 Pfeil, wenn dies erlaubt ist.

+ save (*EasyFileWriter*) → *void*

Speichert diesen Pfeil im EasyFileWriter.

D.9.11 Klasse GrafPunkt

(abstrakte Klasse)
extends *score.GrafSCM.GrafTyp*
implements *score.util.Observer*

Diese Klasse stellt das grafische Äquivalent eines logischen Punktes dar. Ein grafischer Punkt hat eine linke obere Koordinate und eine Größe.

Attribute

lp:*LogPunkt* – Hält das logische Gegenstück zu diesem grafischen Punkt.

size:*Dimension.Double* – Gibt die Größe dieses grafischen Punktes an.

superState:*GrafZustand* – der Oberzustand dieses Punktes

ownOffset:*Point2D.Double* – der eigene Offset bezüglich des Oberzustands

–$ sect:*int* `CONST` – Gibt an, wie breit der Anteil des Punktes ist, der zum Rand gerechnet wird. Ein Wert von `10` heißt dabei, dass ein Zehntel des Zustands gewählt ist.

Konstruktoren

GrafPunkt (*Dimension.Double*)

Erschafft einen grafischen Punkt, der die übergebene Größe hat.

Methoden

+ update (*Observable, Object*) → *void*

Löst ein Refresh aus.

+ replaceObservable (*Observable, Observable*) → *boolean*

Implementation des Observer-Interfaces.

+$ getGrafPunkt (*int*) → *GrafPunkt*

Gibt den grafischen Punkt mit dem angegebenen Identifier zurück.

+$ getGrafPunkt (*String*) → *GrafPunkt*

Gibt den grafischen Punkt mit dem angegebenen Namen zurück. Ist kein solcher Punkt vorhanden, oder ist der Name nicht eindeutig, so wird `Null` zurückgegeben.

+$ getAll () → *Collection*

Gibt eine Collection aller grafischen Punkte zurück.

+$ swapNames (*GrafPunkt, GrafPunkt*) → *void*
Vertauscht die Namen von zwei GrafPunkten

\+ setSuper (*GrafZustand, Point2D.Double*) → *void*
Setzt den Oberzustand und den Offset.

\+ getSuper () → *GrafZustand*
Gibt den Oberzustand zurück.

\+ setOffset (*Point2D.Double*) → *void*
Setzt den Offset dieses Punktes.

\+ getOwnOffset () → *Point2D.Double*
Gibt den Offset dieses Punktes bezüglich seines Oberzustandes wieder.

\# setLogic (*LogPunkt*) → *void*
Setzt das logische Äquivalent dieses Punktes. Hierbei erhält auch der grafische Punkt seinen Identifier, der dem logischen entspricht.

\+ getLogic () → *LogPunkt*
Gibt das logische Äquivalent dieses grafischen Punktes zurück.

\+ renewIdent () → *void*
Stellt sicher, dass dieser Grafische Punkt unter seinem Logischen Identifier zu finden ist.

\+ setName (*String*) → *boolean*
Setzt den Namen dieses Punktes. Gibt `true` zurück, falls eine Änderung stattgefunden hat.

\+ getName () → *String*
Gibt den Namen dieses Punktes zurück. Kann `Null` sein.

\+ getSize () → *Dimension.Double*
Gibt die Größe des Zustandes zuruck.

\+ setSize (*Dimension.Double*) → *void*
Setzt die Größe des grafischen Punktes. Es wird davon ausgegangen, dass diese Größe gültig ist. Mit dieser Methode werden hauptsächlich die Hooks an diesem Zustand umgesetzt.

\+ getCenter () → *Point2D.Double*
Gibt den Mittelpunkt dieses Punktes zurück.

\+ hasSubPoints () → *boolean*
Gibt an, ob dieser Punkt Unterpunkte hat; solange diese Methode nicht überschrieben wird, gibt sie immer `false` zurück.

\+ touches (*Dimension.Double*) → *boolean*
Gibt an, ob dieser Punkt den Rand der alten Größe des Oberzustands berührt.

\+ adoptSize (*Dimension.Double, Dimension.Double*) → *void*
Setzt die neue Größe passend für die übergebene neue Größe des Oberzustands, falls dieser Zustand den Rand des Oberzustands mit der alten Größe berührt.

\+ addIncoming (*GrafPfeil*) → *void*
Fügt eine eingehende Verbindung zu diesem grafischen Punkt hinzu. Das entsprechende wird beim logischen Punkt wiederholt.

\+ addOutgoing (*GrafPfeil*) → *void*
Fügt eine ausgehende Verbindung zu diesem grafischen Punkt hinzu. Das entsprechende wird beim logischen Punkt wiederholt.

\+ getIncoming () → *Vector*
Gibt eine Enumeration der ankommenden Pfeile zurück.

\+ getOutgoing () → *Vector*
Gibt eine Enumeration der abgehenden Pfeile zurück.

\+ hasOutgoing () → *boolean*

Gibt an, ob dieser GrafPunkt ausgehende Verbindungen hat.

+ canExit (*GrafPunkt*) → *boolean*
  Gibt an, ob von diesem Punkt Verbindungen ausgehen, die den übergebenen GrafPunkt verlassen.
+ getAllArrows () → *Vector*
  Gibt einen Vector zurück, der sämtliche ein- und ausgehenden GrafPfeil enthält.
+ removeIncoming (*GrafPfeil*) → *void*
  Entfernt einen ankommenden Pfeil aus diesem Punkt. Das entsprechende wird beim logischen Punkt wiederholt.
+ removeOutgoing (*GrafPfeil*) → *void*
  Entfernt einen ausgehenden Pfeil aus diesem Punkt. Das entsprechende wird beim logishen Punkt wiederholt.
+ redirectArrowsTo (*GrafPunkt*) → *void*
  Lenkt Pfeile, die vorher bei diesem Punkt anfingen oder aufhörten, auf einen anderen Punkt um.
+ addHook (*GrafHook*) → *void*
  Fügt einen Hook zu diesem Punkt hinzu.
+ removeHook (*GrafHook*) → *void*
  Entfernt einen Hook aus diesem Punkt.
+ getPosOfNeighbours (*GrafModellSicht*) → *Vector*
  Gibt die Zeichenkoordinaten der Nachbarknoten dieses Punktes zurück.
+ getPosOfOwnHooks (*GrafModellSicht*) → *Vector*
  Gibt die Zeichenkoordinaten der Hooks zurück, die auf diesem Punkt liegen. Die Reihenfolge entspricht der von getPosOfNeighbours.
+ getPosOfCrossingPairs (*GrafModellSicht*) → *Vector*
  Gibt einen Vector zurück, der in Paaren die Zeichenkoordinaten der Hooks enthält, deren Verbindung die Grenzen dieses States schneidet.
+ getHooksJustOutside () → *Vector*
  Gibt einen Vector von Hooks zurück, die gerade außerhalb dieses GrafPunktes liegen.
+ addHooksJustOutside (*Vector, GrafPunkt*) → *void*
  Fügt zum übergebenen Vector die Hooks hinzu, die gerade außerhalb des übergebenen GrafPunktes liegen.
+ getOwnHooks () → *Vector*
  Gibt alle Hooks zurück, die auf dem Rand dieses Zustands liegen.
isIn (*double, double*) → *boolean*
  Testet die Koordinaten, die bezüglich dieses Punktes angegeben sind. Diese Funktion erzeugt kein neues Objekt.
+ getObjectAt (*double, double, int, double*) → *GrafTyp*
  Gibt den grafischen Punkt zurück, der sich an den angegebenen Koordinaten (relativ zur linken oberen Ecke) befindet. Bei der Suche wird bis in die angegebene Tiefe weitergegangen. Eine Tiefe von `0` heißt, dass nur dieser Zustand betrachtet werden soll. Gibt einen `Null` zurück, wenn kein entsprechendes Objekt gefunden wird.
+ getPointAt (*double, double*) → *GrafPunkt*
  Gibt den Punkt zurück, der sich an dieser Stelle befindet.
+ getPointOrPartitionAt (*double, double*) → *GrafPunkt*
  Gibt den Punkt oder die Partition zurück, die sich an dieser Stelle befindet.
+ getSuperOf (*double, double, double, double*) → *GrafPunkt*
  Gibt den direkten Oberzustand zurück, zu dem dieses Rechteck gehören würde, sprich diesen Zustand, wenn das Rechteck ganz drin liegt. Es wird davon ausgegangen, dass der erste Punkt die linke obere Ecke und der zweite die rechte untere darstellt.
+ makeRelativeToSuper (*double, double, GrafZustand*) → *Point2D.Double*

Gibt einen Punkt zurück, der die übergebenen Koordinaten für den übergebenen Zustand passend macht. Gibt `false` zurück, wenn ein Fehler auftritt, zum Beispiel der angegebene Zustand kein Oberzustand dieses Zustands ist.

+ makeLocalFromSuper (*double, double, GrafZustand*) → *Point2D.Double*

  Gibt einen Punkt zurück, der die übergebenen Koordinaten des übergebenen Zustands für diesen Zustand lokalizisert. Gibt `false` zurück, wenn ein Fehler auftritt, zum Beispiel der angegebene Zustand kein Oberzustand dieses Zustands ist.

+ makeRelativeTo (*double, double, GrafPunkt*) → *Point2D.Double*

  Rechnet lokale Koordinaten in die eines beliebigen anderen Punktes um.

+ isSubPointOf (*GrafPunkt*) → *boolean*

  Gibt zurück, ob dieser Punkt ein Unterpunkt des übergebenen Punktes ist. Dabei zählt nicht nur der direkte Oberzustand. Ist der übergebene Punkt dieser Punkt selbst, wird ebenfalls `true` zurückgegeben.

+ isPartition () → *boolean*

  Gibt zurück, ob dieser Punkt eine Partition ist; das ist nicht der Fall, solange diese Methode nicht überschrieben wird.

+ insertNewState (*double, double, double, double*) → *GrafZustand*

  Fügt einen neuen Zustand ein – muss entsprechend implementiert werden, da in der unüberschriebenen Form keine inneren Koordinaten akzeptiert werden.

+ insertNewConn (*double, double, double, double, int*) → *GrafKonnektor*

  Fügt einen neuen Konnektor unter diesen Punkt ein. Solange diese Methode nicht überschrieben ist, wird keine Bewegung akzeptiert sondern ein InvalidConstructionError geworfen.

+ isValidMove (*GrafPunkt, double, double*) → *boolean*

  Gibt an, ob eine Bewegung gültig ist. Akzeptiert in dieser Form keine Bewegung, die in diesem GrafPunkt aufhört oder diesen schneidet.

+ isValidSize (*Dimension.Double*) → *boolean*

  Gibt an, ob eine neue Größe gültig ist.

+ isValidPartitionSize (*Dimension.Double*) → *boolean*

  Gibt an, ob die Größe für diesen Zustand als Partition gültig ist. Muss entsprechend implementiert werden, da in der unüberschriebenen Form keine Größe akzeptiert wird.

+ movePointIntoThis (*GrafPunkt, double, double*) → *boolean*

  Bewegt den angegebenen Punkt in diesen Zustand hinein. Muss entsprechend implementiert werden, da in der unüberschriebenen Form kein Punkt akzeptiert wird.

+ conflictsWith (*double, double, double, double*) → *boolean*

  Gibt `true` zurück, falls das (in Koordinaten relativ zur linken oberen Ecke) angegebene Rechteck diesen Zustand schneidet. Wenn eines der beiden Rechtecke im anderen liegt, wird `false` (kein Konflikt) zurückgegeben.

+ isWithin (*double, double, double, double*) → *boolean*

  Gibt zurück, ob sich dieser Punkt im angegebenen Rechteck befindet, das in Koordinaten dieses Punktes angegeben wird.

+ draw (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*

  Zeichnet den grafischen Punkt, der angegebene Offset gibt die Position der linken oberen Ecke direkt an. Muss entsprechend implementiert werden.

+ drawArrows (*Painter, GrafModellSicht*) → *void*

  Zeichnet die Pfeiles, die bei diesem Punkt beginnen oder enden.

+ drawArrowsCrossing (*Painter, GrafModellSicht, GrafPunkt*) → *void*

  Zeichnet die Pfeile dieses Punktes, die den übergebenen Punkt verlassen oder betreten.

+ getArrowsCrossing (*GrafPunkt*) → *Vector*

  Gibt einen Vector aller Pfeile zurück, die den übergebenen Punkt verlassen oder betreten.

+ getPartitioners (*double, double, Point2D.Double*) → *Partitioner*

Gibt die möglichen Partitioner für eine Unterteilung an den gegebenen Koordinaten zurück. Muss entsprechend implementiert werden, da in der unüberschriebenen Form keine Partition akzeptiert wird.

+ moveTo (*GrafZustand, Point2D.Double, boolean*) → *void*

  Verschiebt diesen grafischen Punkt zu einem neuen Oberzustand. Dieser muss bereits als logisches Gegenstück einen OR-Zustand besitzen. Der angegebene Offset ist der Offset im neuen Oberzustand.

+ moveTo (*GrafZustand, Point2D.Double*) → *void*

  Bewegt diesen Punkt an die neue Stelle und ruft danach ein notifyObservers auf.

+ copyToNewSuper (*GrafZustand, Hashtable[]*) → *GrafZustand*

  Diese abstrakte Methode muss so implementiert werden, dass sie den GrafPunkt in den neuen Oberzustand kopiert, und dabei die übergebenen Hashtables entsprechend füllt.

+ checkArrowsForCopy (*Hashtable[]*) → *void*

  Füllt die übergebenen Hashtables so, dass alle eingehenden und ausgehenden Pfeile dieses GrafPunkts mit Ereignissen und Variablen beachtet wurden.

+ completeCopy (*Hashtable[], GrafZustand, Vector*) → *void*

  Beendet den Kopiervorgang. Wird diese Methode nicht überschrieben, heißt dies, dass die erforderlichen Pfeile eingefügt werden.

+ translate (*Hashtable[]*) → *GrafPunkt*

  Gibt die Übersetzung dieses grafischen Punktes anhand der übergebenen Hashtables zurück. Ist dieser Punkt nicht in den Hashtables aufgeführt, wird ein Fehler ausgegeben und `Null` zurückgegeben.

+ getDepthTo (*GrafPunkt*) → *int*

  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt an, wie tief dieser Punkt unter dem angegebenen Zustand liegt. Eine Tiefe von `0` heißt, dass es sich um diesen Punkt selbst handelt. Wird der Oberzustand nicht gefunden, bis kein weiterer Oberzustand angegeben ist, wird `-1` zurückgegeben.

+ probeDepth () → *int*

  Gibt die Tiefe zurück, die Zustände unterhalb dieses Zustands haben. Eine Tiefe von `0` gibt an, dass es unter diesem Zustand keine weiteren gibt.

+ getLocalOffset (*double, double, int*) → *Point2D.Double*

  Gibt den Offset zurück, den diese Koordinaten innerhalb dieses Punktes haben, wenn sie innerhalb dieses Punktes liegen.

+ getLocalAlignment (*double, double, int*) → *int*

  Gibt den entsprechenden Wert aus javax.swing.SwingConstants zurück: NORTH, NORHT_EAST, EAST, ... und CENTER. Als illegaler Wert (wenn die übergebeben Koordinaten nicht in diesem Punkt liegen), wird SwingConstants.TRAILING definiert. Wie groß der Anteil des Punktes ist, der zum Rand gehört, wird in der Klassenkonstante `sect` festgelegt.

+ getPosAtBorder (*Point2D.Double*) → *Point2D.Double*

  Gibt die orthogonale Projektion des übergebenen Punktes auf den Rand dieses Punktes zurück. Als Bereich wird hierfür der Wert von getLocalAlignment benutzt.

+ putOntoRim (*Point2D.Double, Point2D.Double*) → *Point2D.Double*

  Gibt die Projektion des ersten Punktes in Richtung des zweiten Punktes auf den Rand dieses grafsichen Punktes zurück.

calcLocAlign (*double, double*) → *int*

  Berechnet den Wert für getLocalAlignment.

+ isSECorner (*double, double, int, double*) → *int*

  Gibt 1 zurück, wenn der Punkt in der Nähe der rechten unteren Ecke liegt, -1 wenn das nicht der Fall ist, und 0 wenn dieser Zustand nicht betroffen ist.

+ delete () → *void*

überschreibt die Methode der Klasse *GrafTyp*.
Löscht diesen GrafPunkt. Dabei werden auch alle ein- und ausgehenden Pfeile beachtet, d. h. es wird von ihnen mindestens ein Abschnitt gelöscht.

+ save (*EasyFileWriter*) → *void*

  Speichert diesen GrafPunkt im EasyFileWriter. Muss entsprechend implementiert werden.

+ saveSelf (*EasyFileWriter*) → *void*

  Speichert die Daten dieses GrafPunkts: Identifier, Name und Größe.

+ saveInsert (*EasyFileWriter*) → *void*

  Schreibt einen Insert-Befehl für diesen GrafPunkt in den EasyFileWriter.

saveTexts (*EasyFileWriter*) → *void*

  Schreibt die Texte für diesen GrafPunkt in den EasyFileWriter.

D.9.12 Klasse GrafSCModell

extends *score.AbstractModell.AbstractModell*
implements *score.util.Observer*

Diese Klasse ist die Sammlung aller grafischen Objekte. Sie beobachtet alle Komponenten und meldet Änderungen an die Sichten weiter.

Konstruktoren

+ GrafSCModell ()

  Der Konstruktor erschafft ein neues grafisches Statechart Modell und ein zugehöriges logisches Statechart Modell. Beide Modelle werden mit einem Oberzustand initialisiert. Dieses Modell wird auch an die grafischen Sichten übergeben.

Methoden

+ replaceBy (*GrafSCModell*) → *void*

  Überschreibt die Methode des Observable, indem der Aufruf an den enthaltenen virtuellen Oberzustand weitergereicht wird.

+ replaceObservable (*Observable, Observable*) → *boolean*

  Implementation des Observer-Interfaces.

+ getLogic () → *LogSCModell*

  Gibt das zugehörige logische Statechart Modell zurück.

+ getTopLevelState () → *GrafTopLevelZust*

  Gibt den virtuellen Oberzustand zurück.

+ getView (*int*) → *GrafModellSicht*

  Gibt eine Sicht auf den angegebenen Zustand zurück. Ist dies kein Zustand, wird `Null` zurückgegeben. Die Sicht wird mit der maximalen Tiefe des Zustands initialisiert.

+ getView (*int, int*) → *GrafModellSicht*

  Gibt eine Sicht auf den angegebenen Zustand zurück, die mit der übergebenen Tiefe initialisiert wird.

+ getViewOfActive () → *GrafModellSicht*

  Gibt eine Sicht auf das momentan aktive Objekt zurück. Ist dies kein Zustand, wird `Null` zurückgegeben.

+ getView () → *GrafModellSicht*

  Gibt eine Sicht auf das gesamte Statechart zurück.

+ getAll () → *Vector*

überschreibt die Methode der Klasse *AbstractModell*.
Gibt einen Vector aller grafischen Punkte und grafischen Pfeile zurück. Der virtuelle Oberzustand ist darin nicht enthalten.

+ editedSinceChecked () → *boolean*

  Gibt an, ob im Grafischen Modell seit der letzten Überpfrüfung Änderungen vorgenommen wurden.

+ editedSinceChecked (*boolean*) → *boolean*

  Setzt den Wert, ob Änderungen seit dem letzten Überprüfen vorgenommen wurden.

+ editedSinceSaved () → *boolean*

  Gibt an, ob im Grafischen Modell seit dem letzten Speichern Änderungen vorgenommen wurden.

+ editedSinceSaved (*boolean*) → *boolean*

  Setzt den Wert, ob Änderungen seit dem letzten Speichern vorgenommen wurden.

+ setEdited () → *void*

  Setzt, dass Änderungen vorgenommen wurden.

+ save (*EasyFileWriter*) → *void*

  Speichert das komplette GrafSCM im EasyFileWriter.

+ update (*Observable, Object*) → *void*

  überschreibt die Methode der Klasse *AbstractModell*.
  Implementation des Observer-Interfaces. Gibt Aufrufe an die Observer des GrafSCModell weiter.

+ alignSize () → *void*

  Setzt die Größe aller markierten Konnektoren auf den selben Wert. Hierfür wird das Minimum aller vorhandenen Größen genommen.

D.9.13 Klasse GrafText

extends *score.GrafSCM.GrafTyp*
Mit GrafText ist es möglich, beliebige Texte in das Statechart einzugeben. Sie können sich entweder in einem Zustand befinden, oder auf einen Pfeil beziehen; im letzteren Fall ist die Position des Textes relativ zu einem bestimmten Hook. Gibt es einen zugehörigen Pfeil, wird auch angegeben, ob der Wert des Textes noch den Eigenschaften des Pfeils entspricht.

Konstruktoren

+ GrafText (*String, GrafTyp, Point2D.Double*)

  Erschafft einen neuen GrafText mit dem angegebenen Inhalt, der sich vom übergebenen GrafTyp, auf den er sich bezieht, an der angegebenen Position befindet.

+ GrafText (*GrafText, GrafTyp*)

  Copy-Konstruktor mit Referenz

Methoden

+ getID () → *int*

  Gibt den Identifier dieses Textes zurück.

+ delete () → *void*

  überschreibt die Methode der Klasse *GrafTyp*.
  Löscht diesen Text aus dem Hashtable aller Texte.

+$ resetCounter () → *void*

  Setzt den Zähler der Texte wieder auf 1 und entfernt alle Texte aus dem Hashtable.

+$ getText (*Integer*) → *GrafText*

  Gibt den Text mit dem angegebenen Identifier zurück.

+$ getAll () → *Collection*

Gibt eine Collection aller Texte zurück.

+$ checkReferences () → *void*

Überprüft bei allen Texten, die sich auf einen Pfeil (bzw. Hook) beziehen, welchen Wert diese Referenz besitzt.

\+ getText () → *String*

Gibt den angezeigten Text zurück.

\+ differs () → *boolean*

Gibt an, ob sich der angezeigte Text von dem der Referenz unterscheidet.

\+ setText (*String*) → *boolean*

Setzt den Text. Bei GrafTexten, die sich auf Pfeile beziehen, kann dadurch eine Abweichung vom Referenztext entstehen. Wird bei solchen Texten der leere Text (oder `Null`) angegeben, so wird der Referenztext gesetzt. Gibt `true` zurück, falls eine Änderung stattgefunden hat.

\+ setLabel (*String*) → *void*

Setzt den Referenztext. Wenn sich der angezeigte Text nicht vom ursprünglichen Referenztext unterschieden hat, wird auch der angezeigte Text geändert.

\+ getReference () → *GrafTyp*

Gibt die Referenz dieses GrafTextes zurück.

\+ setReference (*GrafTyp*) → *void*

Setzt die Referenz dieses GrafTextes.

\+ setReference (*GrafTyp, Point2D.Double*) → *void*

Setzt die Referenz und den Offset dieses GrafTextes.

\+ draw (*Painter, Point2D.Double, GrafModellSicht*) → *void*

Zeichnet diesen GrafText in den übergebenen Painter; der übergebene Offset ist die Position der Referenz.

+$ saveAll (*EasyFileWriter*) → *void*

Speichert alle GrafTexte.

\+ save (*EasyFileWriter*) → *void*

Speichert diesen GrafText.

D.9.14 Klasse GrafTopLevelZust

extends *score.GrafSCM.GrafZustand*
Diese Klasse ist der virtuelle Oberzustand, der schon bei leerer Zeichenfläche vorhanden ist.

Konstruktoren

\+ GrafTopLevelZust ()

Erschafft einen neuen Zustand mit der Größe `0`.

Methoden

\+ getView () → *GrafModellSicht*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt eine neue Sicht auf den Oberzustand zurück. Für diesen Zustand werden die Einstellungen nicht gespeichert.

\# isIn (*double, double*) → *boolean*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt immer `true` zurück, da nach definition jeder Wert in diesem Zustand enthalten ist.

\+ getObjectAt (*double, double, int, double*) → *GrafTyp*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt den grafischen Punkt zurück, der sich an den angegebenen Koordinaten befindet. Bei der Suche wird bis in die angegebene Tiefe weitergegangen. Eine Tiefe von `0` heißt, dass nur die Zustände auf oberster Ebene betrachtet werden. Gibt `Null` zurück, wenn kein entsprechendes Objekt gefunden wird.

\+ draw (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*

überschreibt die Methode der Klasse *GrafZustand*.
Zeichnet diesen grafischen Punkt und Unterzustände bis zur angegebenen Tiefe.

\+ hasSubPoints () → *boolean*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt an, ob dieser Zustand Unterpunkte besitzt.

\+ getPartitioners (*double, double, Point2D.Double*) → *Partitioner*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt die möglichen Partitioner für eine Unterteilung an den gegebenen Koordinaten zurück.

\+ isVirtualTop () → *boolean*

überschreibt die Methode der Klasse *GrafTyp*.
Gibt `true` zurück und zeigt dadurch an, dass dies der virtuelle Oberzustand ist.

\+ getBoundingBox (*Point2D.Double*) → *Dimension.Double*

Gibt die Größe einer umgebenden Box aller Unterzustände zurück. Sind noch keine Unterzustände eingetragen, wird `Null` zurückgegeben.

\+ maybeArrowSource () → *boolean*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt zurück, dass kein Pfeil auf dem virtuellen Oberzustand beginnen kann.

\+ maybeArrowTarget () → *boolean*

überschreibt die Methode der Klasse *GrafZustand*.
Gibt zurück, dass kein Pfeil auf dem virtuellen Oberzustand enden kann.

\+ save (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *GrafZustand*.
Speichert alle Unterpunkte des globalen Wurzelzustands in den EasyFileWriter.

D.9.15 Klasse GrafTyp

extends *score.AbstractModell.MarkableTyp*
Diese Klasse ist ein öffentlicher Idenifier für alle sichtbaren Objekte, die unterschieden werden können. Er enthält die GrafTexte zu diesem Objekt.

Attribute

#$ modell:*AbstractModell* – Das Grafische Statechart Modell, auf das sich die GrafTypen beziehen.

texts:*Vector* – Der Vector, der die GrafTexte enthält, die zu diesem GrafTyp gehören.

Methoden

+$ setModell (*AbstractModell*) → *void*

Setzt das Grafische Statechart Modell.

\+ getDepthTo (*GrafPunkt*) → *int*

Gibt die Tiefe zum übergebenen Grafischen Punkt zurück; wenn diese Methode nicht überschrieben wird, ist das immer `-1`.

\+ isVirtualTop () → *boolean*

Gibt zurück, ob es sich bei diesem GrafTyp um den virtuellen Oberzustand handelt. Wenn diese Methode nicht überschrieben wird, ist das immer `false`.

\+ delete () → *void*

überschreibt die Methode der Klasse *Observable*.
Löscht dieses Objekt.

+ maybeArrowSource () → *boolean*
  Gibt an, ob dieses Objekt Startpunkt eines Pfeils sein darf.
+ maybeArrowTarget () → *boolean*
  Gibt an, ob dieses Objekt Endpunkt eines Pfeils sein darf.
+ addText (*GrafText*) → *void*
  Fügt zu diesem GrafTyp den übergebenen Text hinzu.
+ removeText (*GrafText*) → *void*
  Löscht aus diesem GrafTyp den übergebenen Text.
+ hasTexts () → *boolean*
  Gibt an, ob dieser GrafTyp eigene GrafTexte besitzt.
+ drawTexts (*Painter, Point2D.Double, GrafModellSicht*) → *void*
  Zeichnet die GrafTexte dieses GrafTyps in den übergebenen Painter.

D.9.16 Klasse GrafZustand

extends *score.GrafSCM.GrafPunkt*
Diese Klasse stellt das grafische Äquivalent eines logischen atomaren Zustands dar.

Attribute

subPoints:*Vector* – Hält die Unterzustände dieses Zustands.

Konstruktoren

+ GrafZustand (*Dimension.Double*)
  Erschafft einen grafischen Zustand, der die angegebene Größe hat. Er besitzt noch keine Unterzustände.
+ GrafZustand (*Dimension.Double, ZustandAtom*)
  Erschafft einen neuen grafischen Zustand und initialisiert ihn mit dem übergebenen logischen atomaren Zustand.

Methoden

+ replaceBy (*Observable*) → *void*
  überschreibt die Methode der Klasse *Observable*.
  Ergänzt die Methode replaceBy des Observables durch den entsprechenden Aufruf der Unterpunkte. Ein eventuell vorhandener View erhält ebenfalls einen Aufruf.
+ getView (*int*) → *GrafModellSicht*
  Gibt die Sicht dieses Zustands zurück. Ist noch keine vorhanden, so wird sie jetzt erschaffen, mit diesem Zustand initialisiert und dann zurückgegeben.
  Parameter:
  - int: die Tiefe, die der neue neue View anzeigen soll
+ getView () → *GrafModellSicht*
  Gibt die Sicht dieses Zustands zurück. Ist noch keine vorhanden, so wird sie jetzt erschaffen.
+ hasView () → *boolean*
  Gibt an, ob dieser Zustand eine Sicht besitzt.
+ setView (*GrafModellSicht*) → *void*
  Setzt die Sicht dieses Zustands. Wird `Null` angegeben, hat der Zustand dann keine Sicht mehr.
+ addSubPoint (*GrafPunkt, Point2D.Double*) → *Unterpunkt*

Fügt einen grafischen Unterpunkt hinzu, der bereits eine logische Entsprechung haben muss. Der neue Punkt hat bezüglich dieses Zustands den angegebenen Offset. Ist dieser `Null`, wird `false` zurückgegeben. Der entsprechende Vorgang wird im logischen Modell wiederholt, falls dieser Zustand eine logische Entsprechung hat, die ein OR-Zustand ist. In diesem Fall wird der neue Unterpunkt zurückgegeben, ansonsten `Null`.

\+ addSubConn (*GrafKonnektor, Point2D.Double*) → *Unterpunkt*

Fügt einen neuen Konnektor unterhalb dieses Zustands hinzu.

\+ moveSubPoint (*GrafPunkt, Point2D.Double*) → *Unterpunkt*

Bewegt einen Unterzustand, der aber nach der Bewegung immer noch Unterzustand dieses Punktes ist.

\+ addSubRelative (*GrafPunkt, GrafPunkt, double, double*) → *Unterpunkt*

Fügt einen neuen Zustand hinzu wie addSubPoint, allerdings ist der angegebene Offset in Koordinaten des angegebenen Unterzustands. Ist der übergebene Zustand kein direkter Unterzustand dieses Zustands, wird `Null` zurückgegeben.

\+ removeSubPoint (*GrafPunkt*) → *void*

Entfernt einen Unterzustand. Im logischen Modell wird dieser Vorgang wiederholt.

\# getSubPoint (*GrafPunkt*) → *Unterpunkt*

Gibt den Unterpunkt, der einem GrafPunkt entspricht.

\+ getSubPoints () → *Enumeration*

Gibt eine Enumeration der grafischen Unterpunkte zurück.

\+ probeDepth () → *int*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt die Tiefe zurück, die unterhalb dieses Zustands existiert. Eine Tiefe von `0` gibt an, dass es unterhalb dieses Zustands keine weiteren gibt.

\+ getLocalOffset (*double, double, int*) → *Point2D.Double*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt den Punkt zurück, der die übergebenen Koordinaten auf den untersten Zustand umrechnet, oder `Null`, wenn diese Koordinaten nicht in diesem Punkt liegen, oder es sich um eine Partition handelt.

\+ addPosOfCrossing (*Vector, GrafPunkt, Vector, GrafModellSicht*) → *void*

überschreibt die Methode der Klasse *GrafPunkt*.
Fügt zum übergebenen Vector Paare von Zeichenkoordinaten der Hooks hinzu, deren Verbindungsstrecke die Grenzen dieses Zustands schneidet.

\+ addHooksJustOutside (*Vector, GrafPunkt*) → *void*

überschreibt die Methode der Klasse *GrafPunkt*.
Fügt zum übergebenen Vector die Hooks hinzu, die gerade außerhalb des angegebenen Zustands liegen.

\+ canBeExited () → *boolean*

Gibt an, ob dieser Zustand verlassen werden kann.

\+ canExit (*GrafPunkt*) → *boolean*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt an, ob von diesem Punkt oder seinen Unterpunkten Verbindungen ausgehen, die den übergebenen GrafPunkt verlassen.

\+ getLocalAlignment (*double, double, int*) → *int*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt den entsprechenden Wert aus javax.swing.SwingConstants zurück: NORTH, NORHT_EAST, EAST, ... und CENTER. Als illegaler Wert (wenn die übergebeben Koordinaten nicht in diesem Punkt liegen), wird SwingConstants.TRAILING definiert.

\+ isSECorner (*double, double, int, double*) → *int*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt 1 zurück, wenn der Punkt in der Nähe der rechten unteren Ecke liegt, -1 wenn das nicht der Fall ist, und 0 wenn dieser Zustand nicht betroffen ist.

\+ getObjectAt (*double, double, int, double*) → *GrafTyp*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt den grafischen Punkt zurück, der sich an den angegebenen Koordinaten (relativ zur linken oberen Ecke dieses Zustands) befindet. Bei der Suche wird bis in die angegebene Tiefe weitergegangen. Eine Tiefe von `0` heißt, dass nur dieser Zustand betrachtet werden soll, aber nicht seine Unterzustände oder Pfeile. Gibt `Null` zurück, wenn kein entsprechendes Objekt gefunden wird.

\+ getPointOrPartitionAt (*double, double*) → *GrafPunkt*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt den grafischen Punkt an den angegebenen Koordianten zurück.

\+ draw (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*

überschreibt die Methode der Klasse *GrafPunkt*.
Zeichnet diesen grafischen Punkt und Unterzustände bis zur angegebenen Tiefe. Eine Tiefe von `0` heißt dabei, dass nur dieser Zustand gezeichnet werden soll. Der angegebene Offset ist der Punkt, an dem die linke obere Ecke dieses Zustands effektiv gezeichnet werden soll.

\+ drawArrowsCrossing (*Painter, GrafModellSicht, GrafPunkt*) → *void*

überschreibt die Methode der Klasse *GrafPunkt*.
Zeichnet die Pfeile, die den übergebenen Zustand verlassen oder betreten.

\+ getArrowsCrossing (*GrafPunkt*) → *Vector*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt einen Vector aller Pfeile zurück, die den übergebenen Zustand verlassen oder betreten.

\# drawContents (*int, Painter, Point2D.Double, GrafModellSicht*) → *void*

Zeichnet Unterzustände und enthaltene Pfeile.

\+ hasSubPoints () → *boolean*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt an, ob dieser Zustand Unterpunkte besitzt.

\+ hasSubStates () → *boolean*

Gibt an, ob dieser Zustand Unterzustände besitzt.

\+ getSuperOf (*double, double, double, double*) → *GrafPunkt*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt den direkten Oberzustand zurück, zu dem dieses Rechteck gehören würde, der aber gleichzeitig Unterzustand dieses Zustandes (oder dieser Zustand selbst) ist. Wenn es keinen solchen eindeutigen Zustand gibt, wird `Null` zurückgegeben. Schneidet das Rechteck vorhandene Unterzustände, wird ein InvalidConstructionError geworfen. Diese Methode geht davon aus, dass dass der erste Punkt die linke obere Ecke und der zweite die rechte untere Ecke des Rechtecks angibt, in Koordinaten dieses Zustands. Es wird garantiert, dass ein Zustand (oder `Null`) zurückgegeben wird.

\+ insertNewState (*double, double, double, double*) → *GrafZustand*

überschreibt die Methode der Klasse *GrafPunkt*.
Fügt einen neuen grafischen Zustand an der gegebenen Stelle ein. Die Koordinaten sind relativ zu diesem Zustand. Der neue Zustand darf keine alten Zustände schneiden, er darf aber von Anfang an Unterzustände enthalten. Diese werden ihm dann gleich untergeordnet. Je nach dem wird ihm entweder ein logischer atomarer Zustand oder ein logischer OR-Zustand zugeordnet. Gibt den neuen Zustand zurück, oder `Null`, wenn die Koordinaten keinen Unterzustand dieses Zustandes beschreiben.

\+ insertNewConn (*double, double, double, double, int*) → *GrafKonnektor*

überschreibt die Methode der Klasse *GrafPunkt*.
Fügt einen neuen Konnektor vom angegebenen Typ ein.

\+ isValidMove (*GrafPunkt, double, double*) → *boolean*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt zurück, ob der angegebene GrafPunkt an den angegebenen Platz (in Koordinaten dieses Punktes) passt. Gibt `false` zurück, wenn dieser Zustand von der Frage nicht betroffen ist oder selbst der bewegte GrafPunkt ist, und `true`, wenn das neue Rechteck komplett in diesem Zustand oder einem Unterzustand liegt. Wenn das neue Rechteck diesen Zustand oder einen Unterzustand schneidet oder überdeckt, wird ein InvalidConstructionError geworfen.

+ isValidSize (*Dimension.Double*) → *boolean*
überschreibt die Methode der Klasse *GrafPunkt*.
Gibt zurück, ob die angegebene Größe für diesen Zustand gültig ist.

+ isValidPartitionSize (*Dimension.Double*) → *boolean*
überschreibt die Methode der Klasse *GrafPunkt*.
Testet, ob der Zustand durch die neue Größe Unterzustände verliert.

+ isValidHMove (*double, double*) → *boolean*
Testet, ob die horizontale Partitionsgrenzenverschiebung von fromX nach toX gültig ist.

+ isValidVMove (*double, double*) → *boolean*
Testet, ob die vertikale Partitionsgrenzenverschiebung von fromY nach toY gültig ist.

+ doHMove (*double, double, double, double*) → *void*
Führt eine horizontale Partitionsgrenzenverschiebung durch. Es wird davon ausgegangen, dass diese Verschiebung gültig ist.
Parameter:

  – fromX: X-Koordinate des Starts
  – toX: X-Koordinate des Ziels

+ doVMove (*double, double, double, double*) → *void*
Führt eine vertikale Partitionsgrenzenverschiebung durch. Es wird davon ausgegangen, dass diese Verschiebung gültig ist.
Parameter:

  – fromY: Y-Koordinate des Starts
  – toY: T-Koordinate des Ziels

+ containsOnlyThis (*GrafPunkt, Point2D.Double, Dimension.Double*) → *boolean*
Gibt zurück, ob das durch offset und newSize angegebene Rechteck nur den übergebenen Punkt als direkten Unterzustand enthält und nicht mit diesem Zustand in Konflikt steht.

+ movePointIntoThis (*GrafPunkt, double, double*) → *boolean*
überschreibt die Methode der Klasse *GrafPunkt*.
Bewegt den Punkt in diesen Zustand (oder einen Unterzustand). Gibt `true` zurück, falls eine Bewegung stattgefunden hat, ansonsten `false`.

+ getDefaultEntry () → *GrafKonnektor*
Gibt den Default-Entry-Konnektor zurück, der für diesen Zustand gesetzt ist. Wenn keiner gesetzt ist, wird `Null` zurückgegeben. Sind mehrere Default-Entry-Konnektoren in diesen Punkt eingezeichnet, wird der erste zurückgegeben, der gefunden wird.

+ hasUniqueDefEntry () → *boolean*
Gibt `true` zurück, wenn in diesen Punkt genau ein Default-Entry-Konnektor eingetragen ist.

+ hasHistoryEntry () → *boolean*
Gibt zurück, ob dieser Zustand einen FlatHistory-Konnektor enthält.

+ getPartitioners (*double, double, Point2D.Double*) → *Partitioner*
überschreibt die Methode der Klasse *GrafPunkt*.
Erhält einen Punkt in eigenen Koordinaten, an dem eine Partitionierung getestet werden soll. Der angegebene Offset ist der Zeichenoffset dieses Zustandes. Gibt einen zweistelligen Array von Partitioners zurück, wobei der erste der horizontale und der zweite der vertikale ist. Ist eine von beiden Richtungen (oder beide) ungültig, so wird an Stelle des entsprechenden Partitioners `Null` zurückgegeben. Liegt der Punkt überhaupt nicht in diesem Zustand, wird `Null` zurückgegeben.

+ isPartition () → *boolean*

überschreibt die Methode der Klasse *GrafPunkt*.
Gibt zurück, ob dieser Zustand eine Partition eines AND-Zustands ist.

+ partitionHorizAt (*double*) → *GrafZustand*

  Teilt diesen Zustand, der eine Partition sein muss, horizontal an der angegebenen Koordinate (bzgl. dieses Zustands). Es wird ein neuer grafischer Zustand erzeugt, dessen logisches Gegenstück ein OR-Zustand ist, und die Unterzustände, die unterhalb der Koordinate liegen, werden in diesen eingefügt. Der neue Zustand wird zurückgegeben, und in den AND-Oberzustand eingefügt. Ist dieser Zustand keine Partition, wird `Null` zurückgegeben.

+ partitionVertAt (*double*) → *GrafZustand*

  Teilt diesen Zustand, der eine Partition sein muss, vertikal an der angegebenen Koordinate (bzgl. dieses Zustands). Es wird ein neuer grafischer Zustand erzeugt, dessen logisches Gegenstück ein OR-Zustand ist, und die Unterzustände, die unterhalb der Koordinate liegen, werden in diesen eingefügt. Der neue Zustand wird zurückgegeben, und in den AND-Oberzustand eingefügt. Ist dieser Zustand keine Partition, wird `Null` zurückgegeben.

+ copyToNewSuper (*GrafZustand, Hashtable[]*) → *GrafZustand*

  überschreibt die Methode der Klasse *GrafPunkt*.
  Kopiert diesen Zustand in den übergebenen neuen Oberzustand. Größe und Offset dieses Zustands werden dabei übernommen. In translators stehen die Übertragungstabellen für Zustände, Ereignisse und Variablen.

+ copyToNewSuper (*GrafZustand, Point2D.Double, Hashtable[]*) → *GrafZustand*

  Kopiert diesen Zustand, wobei der Offset im neuen Oberzustand explizit angegeben wird.

+ completeCopy (*Hashtable[], GrafZustand, Vector*) → *void*

  überschreibt die Methode der Klasse *GrafPunkt*.
  Schließt den Kopiervorgang ab. Dabei werden insbesondere die Aktionen beim Aktivieren und Deaktivieren gesetzt.

+ delete () → *void*

  überschreibt die Methode der Klasse *GrafPunkt*.
  Löscht diesen Zustand und bewegt eventuell vorhandene Unterzustande in den Oberzustand des gelöschten OR-Zustands.

+ maybeArrowSource () → *boolean*

  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt an, dass dieser Zustand Startpunkt eines Pfeils sein darf.

+ maybeArrowTarget () → *boolean*

  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt an, dass dieser Zustand Endpunkt eines Pfeils sein darf.

+ save (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *GrafPunkt*.
  Speichert diesen Zustand im EasyFileWriter.

D.9.17 Klasse Partitioner

extends *score.GrafSCM.GrafTyp*
Diese Klasse stellt eine Partitions-Grenzlinie dar. Sie hat einen Verweis auf den Zustand, den sie partitioniert und zwei Endpunkte in Koordinaten dieses Zustands.

Attribute

– superOffset:*Point2D.Double* – Diese Variable enthält den Offset des Oberzustands bezüglich des Hauptzustands, in dem dieser Partitioner eingefügt werden soll. Ist `Null`, sobald diese Partition wirklich durchgeführt wurde.

Konstruktoren

+ Partitioner (*Point2D.Double, Point2D.Double, GrafZustand, Point2D.Double*)

Erzeugt einen neuen Partitioner, der aber zunächst nicht eingefügt ist, sondern nur eine Möglichkeit darstellt. Er liegt zwischen den beiden Punkten, die bezüglich des Oberzustandes angegeben werden, den dieser Partitioner teilen würde. Es wird davon ausgegangen, dass diese Punkte in der Reihenfolge links-rechts bzw. oben-unten angegeben sind. Der angegebene Offset ist der Zeichenoffset des Oberzustandes in der Sicht, in der gerade gearbeitet wird.

Methoden

+ getSuper () → *GrafZustand*

  Gibt den Zustand zurück, in dem dieser Partitioner installiert ist.

+ draw (*Painter*) → *boolean*

  Zeichnet den Partitioner in den Painter; es wird davon ausgegangen, dass der Partitioner noch nicht installiert ist, und es wird der intern gemerkte Offset bentuzt. Ist dieser `Null`, wird `false` zurückgegeben, ansonsten `true`.

+ draw (*Painter, Point2D.Double*) → *boolean*

  Zeichnet den Partitioner in den Painter, wobei der übergebene totale Offset benutzt wird. Ist der Partitioner noch nicht installiert, wird `false` zurückgegeben, ansonsten `true`.

+ isIn (*double, double, double*) → *boolean*

  Gibt zurück, ob die übergebenen Koordinaten auf diesem Partitioner liegen. Ist der Partitioner noch nicht installiert, wird `false` zurückgegeben.

+ isIn (*double, double*) → *boolean*

  Gibt zurück, ob die angegebenen Koordinaten direkt auf dem Partitioner liegen.

+ getObjectAt (*double, double, double*) → *GrafTyp*

  Gibt diesen Partitioner zurück, falls die Koordinaten auf ihm liegen.

+ install (*GrafModellSicht*) → *int*

  Installiert diesen Partitioner in seinen Oberzustand. Gibt `-1` zurück, wenn kein Oberzustand gesetzt ist oder der Partitioner bereits installiert wurde, ansonsten die Tiefe des Partitioners unter dem Oberzustand der angegebenen Sicht. Am Ende dieser Methode wird ein Refresh des Modells aufgerufen.

+ isHorizontal () → *boolean*

  Gibt zurück, ob dieser Partitioner waagrecht liegt.

+ isVertical () → *boolean*

  Gibt zurück, ob dieser Partitioner senkrecht liegt.

+ isInstalled () → *boolean*

  Gibt zurück, ob dieser Partitioner schon installiert wurde.

+ getUninstalledFirst () → *Point2D.Double*

  Gibt den ersten Endpunkt des Partitioners zurück. Es wird davon ausgegangen, dass der Partitioner noch nicht installiert ist.

+ getUninstalledSecond () → *Point2D.Double*

  Gibt den zweiten Endpunkt des Partitioners zurück. Es wird davon ausgegangen, dass der Partitioner noch nicht installiert ist.

+ getInstalledFirst (*GrafModellSicht*) → *Point2D.Double*

  Gibt den Anfangspunkt bezüglich des Zustands zurück, in dem dieser Partitioner installiert ist.

+ getInstalledSecond (*GrafModellSicht*) → *Point2D.Double*

  Gibt den Endpunkt bezüglich des Zustands zurück, in dem dieser Partitioner installiert ist.

+ getDepthTo (*GrafPunkt*) → *int*

  überschreibt die Methode der Klasse *GrafTyp*.
  Gibt die Tiefe des Partitioners zum gegebenen Oberzustand zurück. Diese ist um 2 größer als die Tiefe des zugehörigen AND-Zustands. Wird kein Oberzustand gefunden, wird `-1` zurückgegeben.

+ touches (*Dimension.Double*) → *boolean*

Gibt an, ob der Partitioner den Rand des Oberzustands mit der übergebenen Größe berührt.

+ adoptSize (*Dimension.Double*) → *void*

  Setzt die Größe so, dass sie zur übergebeben neuen Größe des Oberzustands passt.

+ isValidMove (*Point2D.Double*) → *boolean*

  Gibt zurück, ob der übergebenen Punkt eine gültige Bewegung angibt. Dabei wird (je nach Ausrichtung des Partitioners) nur die X- bzw. Y-Koordiante beachtet.

+ moveTo (*Point2D.Double*) → *void*

  Bewegt diesen Partitioner an die angegebene Position. Dabei wird (je nach Ausrichtung dieses Partitioners) nur die X- bzw. Y-Koordinate beachtet.

+ resizeH (*double, double, double, double*) → *void*

  Verlängert oder verkürzt diesen Partitioner, wenn er durch die angegebene Bewegung betroffen ist.

+ resizeV (*double, double, double, double*) → *void*

  Verlängert oder verkürzt diesen Partitioner, wenn er durch die angegebene Bewegung betroffen ist.

+ save (*EasyFileWriter*) → *void*

  Speichert diesen Partitioner im EasyFileWriter.

D.9.18 Klasse SuperList

extends *java.lang.Object*
Diese Klasse stellt die Liste der Oberzustände eines Zustands dar und ermöglicht es, den tiefsten gemeinsamen Oberzustand zu finden.

Konstruktoren

+ SuperList (*GrafPunkt*)

  Erschafft einen neue Liste, die durch die Oberzustände des übergebenen Punktes initialisiert wird.

+ SuperList (*SuperList*)

  Kopiert den noch zu beachtenden Teil einer SuperList.

Methoden

+ consider (*GrafPunkt*) → *int*

  Betrachtet zusätzlich einen weiteren neuen Grafischen Punkt. Gibt die Position des tiefsten gemeinsamen Oberzustands in dieser Liste zurück.

+ getFirstFor (*GrafPunkt*) → *GrafPunkt*

  Gibt den tiefsten gemeinsamen Oberzustand aller betrachteten Punkte zuzüglich dieses Punktes zurück.

+ getFirstCommon () → *GrafPunkt*

  Gibt den tiefsten gemeinsamen Oberzustand aller betrachteten grafischen Punkte zurück.

D.9.19 Klasse Unterpunkt

extends *java.lang.Object*
Diese Klasse ist ein Paar aus einem Offset und einem Grafischen Punkt.

Attribute

+ offset:*Point2D.Double* – Der Offset des Unterpunktes.
+ point:*GrafPunkt* – Der grafische Punkt, der an der entsprechenden Stelle liegt.

Konstruktoren

+ Unterpunkt (*GrafPunkt, Point2D.Double*)

Erschafft einen neuen Unterpunkt mit dem entsprechenden Punkt und übergebenen Offset.

Methoden

+ localizeX (*double*) → *double*

  Transformiert die X-Koordinate des Oberzustands in lokale Koordinaten dieses Unterzustands.

+ localizeY (*double*) → *double*

  Transformiert die Y-Koordinate des Oberzustands in lokale Koordinaten dieses Unterzustands.

+ getTotalOffset (*Point2D.Double*) → *Point2D.Double*

  Gibt den Punkt zurück, an dem die linke obere Ecke dieses Unterpunktes effektiv gezeichnet werden soll. Genauer gesagt wird der Offset dieses Unterpunktes zum übergebenen Offset addiert und das Ergebnis zurückgegeben.

D.10 Package score.LogDM

D.10.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.AbstractModell.AbstractModell*
    ↪ *score.LogDM.LogDModell*
  ↪ *score.AbstractModell.MarkableTyp*
    ↪ *score.LogDM.DlgKomponente*
      ↪ *score.LogDM.DlgButton*
      ↪ *score.LogDM.DlgCheckbox*
      ↪ *score.LogDM.DlgPopup*
      ↪ *score.LogDM.DlgRadiobutton*
      ↪ *score.LogDM.DlgText*
      ↪ *score.LogDM.DlgTimer*

D.10.2 Interface DialogCompManager

Regelt den Zugriff auf eine Dialogkomponente

Methoden

+ setDlgVisible (*int*, *boolean*) → *void*

  Macht die Dialogkomponente id visible oder invisible

+ setOn (*int*) → *void*

  Setzt Komponente id auf on

+ setOff (*int*) → *void*

  Setzt Komponente id auf off

D.10.3 Interface DialogCompServer

Regelt den Zugriff um eine Dialogkomponente zu erhalten

Methoden

+ getComponentWithId (*int*) → *Object*

  Soll die Komponente mit der ID id zurückgeben

D.10.4 Klasse DlgButton

extends *score.LogDM.DlgKomponente*
Ein Button

Konstruktoren

+ DlgButton ()
  Konstruktor mit DefaultAktion
+ DlgButton (*String*)
  Konstruktor, setz den Label

Methoden

+ setVisible (*boolean*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt sichtbar oder unsichtbar
+ setOn () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  wird ignoriert
+ setOff () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  wird ignoriert
+ toString () → *String*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Liefert eine String-Repräsentation
+ createComponent () → *Component*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Erstellt eine grafische Komponente
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Speichert den Button
+ setLabel (*String*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt den Label des Buttons
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Exportiert den Quelltext

D.10.5 Klasse DlgCheckbox

extends *score.LogDM.DlgKomponente*
Eine Checkbox

Konstruktoren

+ DlgCheckbox ()
  Kontruktor mit DefaultAktion
+ DlgCheckbox (*String*)
  Konstruktor, setzt den Label

Methoden

+ setVisible (*boolean*) → *void*

überschreibt die Methode der Klasse *DlgKomponente*.
Setzt sichtbar oder unsichtbar

+ setOn () → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Setzt die Checkbox auf selected

+ setOff () → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Setzt die Checkbox auf deselected

+ isSelected () → *boolean*
Prüft ob die Checkbox selected ist

+ toString () → *String*
überschreibt die Methode der Klasse *DlgKomponente*.
Gibt einen String mit CHECKBOX ID und Label aus

+ createComponent () → *Component*
überschreibt die Methode der Klasse *DlgKomponente*.
Erzeugt eine Java-Checkbox

+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Speichert die Checkbox

+ setLabel (*String*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Setzt den Label

+ exportQT (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Exportiert den zugehörigen Quelltext

D.10.6 Klasse DlgKomponente

(abstrakte Klasse)
extends *score.AbstractModell.MarkableTyp*
implements *score.Aktionen.Actor,*
java.awt.event.ActionListener

Abstrakte Oberklasse der Dialogkomponenten

Attribute

+ user:*boolean* – spezifiziert ob user(true) oder systemdialog(false) Komponente

visible:*boolean* – spezifiziert momentan ob sichtbar oder unsichtbar

defVisible:*boolean* – Spezifiziert ob Defaultmäßig sichtbar oder unsichtbar

#$ number:*int* – Statischer Zähler der Dialogkomponenten

actions:*AktKomplex* – Der Dialogkomponenten zugeordnete Aktionen

generatedActions:*AktKomplex* – Während der Simulation generierte Aktionen

Konstruktoren

+ DlgKomponente ()
Konstruktor Erhöht Komponentenzähler und setzt id

Methoden

+$ setCounter (*int*) → *void*
Setzt den globalen Komponenten Counter

+ setID (*int*) → *void*

Setzt die id dieser Komponente auf i

+ getID () → *int*
  Liefert die ID dieser Komponente
+ setLabel (*String*) → *void*
  Setzt den Label dieser Komponente
  Parameter:
  – s: neuer Label
+ getLabel () → *String*
  Liefert den Label dieser Komponente
+ setFromVar (*String*) → *void*
  Setzt den Wert von einer Variablen
+ setVisible (*boolean*) → *void*
  Macht die Komponenten sichtbar oder unsichtbar
  Parameter:
  – visible: wenn true, setze sichtbar, sonst sunsichtbar
+ setDefaultVisible (*boolean*) → *void*
  Setzt diese Defaultmäßig auf visible
+ isVisible () → *boolean*
  Prüft ob die Komponenten momentan sichtbar ist
+ isDefaultVisible () → *boolean*
  Prüft ob die Komponente Defaultmäßig sichtbar ist
+ setOn () → *void*
  Die Behandlung einer setOn Aktion
+ setOff () → *void*
  Die Behandlung einer setOff Aktion
+ createComponent () → *Component*
  Erzeugt die Java-Component
+ toString () → *String*
  überschreibt die Methode der Klasse *MarkableTyp*.
  Liefert eine String-Identifikation
+ getActorType () → *int*
  Implementation für Actor-Interface liefert ACTOR_DIALOG
+ getCalledAction (*int*) → *CalledAction*
  Liefert die Aktionen die zugeordnet sind wenn die mehrere sind, wird i ausgewertet
  Parameter:
  – i: welche Aktion?
+ getCalledAction () → *CalledAction*
  Liefert die erste zugeordnete Aktion
+ removeAction (*Aktion*) → *void*
  Löscht eine Aktion
+ doesAction (*Aktion*) → *boolean*
  Prüft ob die übergebene Aktion von dieser Komponente erzeugt wird
  Parameter:
  – action: zu prüfende Aktion
+ getCalledActions () → *AktKomplex*

Liefert die während der Simulation entstandenen Aktionen

+ getActions () → *AktKomplex*
Liefert die dieser Komponenten zugeordneten Aktionen

throwAktionen () → *void*
Interner Aufruf das die Aktionen geworfen werden sollen

+ actionPerformed (*ActionEvent*) → *void*
implementiert das ActionListener-Interface für die Komponenten

+ save (*EasyFileWriter*) → *void*
Speichert die Komponente

+ exportQT (*EasyFileWriter*) → *void*
Erzeugt den Quelltext dieser Komponenten

+ delete () → *void*
überschreibt die Methode der Klasse *Observable*.
Löscht eine Dialogkomponente

D.10.7 Klasse DlgPopup

extends *score.LogDM.DlgKomponente*
Ein Popup

Konstruktoren

+ DlgPopup ()
Erzeugt einen neuen DlgPopup

+ DlgPopup (*String*)
Erzeugt einen neuen DlgPopup mit Label lab

Methoden

+ setVisible (*boolean*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Macht das Popup sichtbar

+ setOn () → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Keine Funktion hier

+ setOff () → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Keine Funktion hier

+ toString () → *String*
überschreibt die Methode der Klasse *DlgKomponente*.
Gibt einen String POPUP+ID und Label aus

+ createComponent () → *Component*
überschreibt die Methode der Klasse *DlgKomponente*.
Erzeugt ein Äquivalent vom Typ Component

+ save (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Speichert das Popup

+ setText (*String*) → *void*
Setzt den Label

+ setFromVar (*String*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Setzt den Label mit einem Variablen Wert

+ exportQT (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *DlgKomponente*.
Erzeugt den zugehörigen Quelltext

D.10.8 Klasse DlgRadiobutton

extends *score.LogDM.DlgKomponente*
Ein Radiobutton

Konstruktoren

+ DlgRadiobutton ()
  Kontruktor mit DefaultAktion
+ DlgRadiobutton (*String*)
  Konstruktor, setzt den Label

Methoden

+ setVisible (*boolean*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt sichtbar oder unsichtbar
+ setOn () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt diesen Radiobutton auf angewählt
+ setOff () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt diesen Radiobutton auf nicht angewählt
+ toString () → *String*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Liefert eine String-Repräsentation
+ createComponent () → *Component*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Erzeugt einen JRadioButton
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Speichert diese Komponente
+ setLabel (*String*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt den Label des Radiobuttons
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Exportiert den Quelltext des Radiobuttons

D.10.9 Klasse DlgText

extends *score.LogDM.DlgKomponente*
Ein Textfeld

Attribute

+$ varserver:*VarServer* – Eine Referenz auf den VarServer

Konstruktoren

+ DlgText ()
  Konstruktor mit DefaultAktion
+ DlgText (*String*)

Konstruktor, setzt den Label

Methoden

+ setVisible (*boolean*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt sichtbar oder unsichtbar
+ setOn () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  wird ignoriert
+ setOff () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  wird ignoriert
+ toString () → *String*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Liefert die String-Repräsentation
+ createComponent () → *Component*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Erzeugt eine grafische Komponente
+ setText (*String*) → *void*
  setzt den Text des Feldes
+ setFromVar (*String*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt dieses Textfeld von einer Variablen
+ actionPerformed (*ActionEvent*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Reagiert auf eine Textfeldänderung (mit Return) und setzt die zugehörige Variable
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Speichert das Textfeld
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Exportiert den Quelltext zum Textfeld

D.10.10 Klasse DlgTimer

extends *score.LogDM.DlgKomponente*
implements *java.lang.Runnable*

Die Dialogkomponente Timer

Konstruktoren

+ DlgTimer ()
  Konstruktor mit DefaultAktion
+ DlgTimer (*String*)
  Konstruktor, setzt den Label

Methoden

+ setVisible (*boolean*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt sichtbar oder unsichtbar
+ toString () → *String*

überschreibt die Methode der Klasse *DlgKomponente*.
Liefert die String-Identifikation

+ createComponent () → *Component*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Erzeugt eine grafische Komponente des Timers
+ setValue (*int*) → *void*
  Setzt den Timer
  Parameter:
  – val: Wert auf den Timer gesetzt wird.
+ setFromVar (*String*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Setzt den Wert von einer Variablen
+ setOn () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Startet den Timer
+ setOff () → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Hält den Timer an
+ run () → *void*
  Für das Runnable Interface – läßt den Timer laufen
+ actionPerformed (*ActionEvent*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Für den Button in der Simulation – ActionListener
+ save (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Speichert den Timer
+ exportQT (*EasyFileWriter*) → *void*
  überschreibt die Methode der Klasse *DlgKomponente*.
  Exportiert den Quelltext

D.10.11 Klasse LogDModell

extends *score.AbstractModell.AbstractModell*
implements *score.LogDM.DialogCompServer*

Das logische Dialogmodell

Attribute

– komp:*Vector* – Vector, in dem alle Komponenten gehalten werden

Konstruktoren

+ LogDModell ()
  Konstruktor Erzeugt einen neuen Komponentenvektor und setzt die DlgKomponenten.ID auf 0

Methoden

+ add (*DlgKomponente*) → *void*
  Eine neue Dialogkomponente hinzufügen
  Parameter:
  – k: Die Komponenten die hinzugefügt werden soll
+ delete (*DlgKomponente*) → *void*

Löscht eine Komponente aus dem Modell

+ replaceObservable (*Observable, Observable*) → *boolean*
Dummy

+ replaceBy (*Observable*) → *void*
überschreibt die Methode der Klasse *Observable*.
Das Modell wird ersetzt

+ getEnumeration () → *Enumeration*
Rückgabewert: Enumeration der Komponenten
Liefert eine Enumeration aller Komponenten

+ getComponentWithId (*int*) → *Object*
Liefert zu einer ID eine Komponente

+ getMaxID () → *int*
Liefert die größte bisher vergeben ID

+ get (*int*) → *DlgKomponente*
Rückgabewert: Dialogkomponente an der i-ten Stelle
Liefert die Komponente an der i-ten Position

+ getAll () → *Vector*
überschreibt die Methode der Klasse *AbstractModell*.
Liefert einen Vektor aller Komponenten

+ addButton () → *DlgButton*
Fügt einen neuen Button hinzu

+ addTextfield () → *DlgText*
Fügt ein neues Textfeld hinzu

+ addRadiobutton () → *DlgRadiobutton*
Fügt einen neuen Radiobutton hinzu

+ createCheckboxes (*int*) → *Vector*
Erzeugt n Checkboxes

+ createRadiobuttons (*int*) → *Vector*
Erzeugt n Radiobuttons

+ addCheckbox () → *DlgCheckbox*
Fügt eine neue Checkbox hinzu

+ addTimer () → *DlgTimer*
Fügt einen neuen Timer hinzu

+ addPopup () → *DlgPopup*
Fügt einen neuen Timer hinzu

+ save (*EasyFileWriter*) → *void*
Speichert das komplette GrafSCM im EasyFileWriter.

+ exportQT (*EasyFileWriter*) → *void*
Erzeugt den Quelltext des logischen Dialogmodells

D.11 Package score.LogSCM

D.11.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.util.Observable*
    ↪ *score.LogSCM.LogComponent*
      ↪ *score.LogSCM.LogPfeil*

↪ *score.LogSCM.LogPfeil_1_1*
↪ *score.LogSCM.LogPfeil_1_N*
↪ *score.LogSCM.LogPfeil_N_1*
↪ *score.LogSCM.LogPunkt*
↪ *score.LogSCM.Konnektor*
↪ *score.LogSCM.KonDefault*
↪ *score.LogSCM.KonHistory*
↪ *score.LogSCM.KonUniversal*
↪ *score.LogSCM.ZustandAtom*
↪ *score.LogSCM.ZustKomplex*
↪ *score.LogSCM.ZustAND*
↪ *score.LogSCM.ZustOR*
↪ *score.LogSCM.LogSCModell*

D.11.2 Klasse KonDefault

extends *score.LogSCM.Konnektor*
Diese Klasse konkretisiert den Konnektor als Default-Einstiegs-Konnektor. Sie dient lediglich der Vereinfachung des Editors, da der wirkliche Default-Einstieg im logischen Modell anders realisiert ist.

Konstruktoren

+ KonDefault ()

  Der Konstruktor setzt den Typ dieses Objektes.

D.11.3 Klasse KonHistory

extends *score.LogSCM.Konnektor*
Diese Klasse konkretisiert den Konnektor als History-Konnektor.

Konstruktoren

+ KonHistory ()

  Der Konstruktor erzeugt einen History-Konnektor. Er wird anfangs nicht als Deep-History-Konnektor gesetzt.

+ KonHistory (*boolean*)

  Erschafft einen neuen History-Konnektor, der als Deep-History-Konnektor gesetzt wird, wenn `true` übergeben wird.

Methoden

+ setDeep (*boolean*) → *void*

  Setzt die Angabe, ob es sich bei diesem Konnektor um einen Deep-History-Konnektor handelt oder nicht.

+ exportQT (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *LogPunkt*.
  Exportiert diesen Historykonnektor in den EasyFileWriter.

D.11.4 Klasse Konnektor

(abstrakte Klasse)
extends *score.LogSCM.LogPunkt*
Diese abstrakte Klasse liegt zwischen einem logischen Punkt und den beiden konkreten Konnektor-Klassen.

D.11.5 Klasse KonUniversal

extends *score.LogSCM.Konnektor*
Diese Klasse konkretisiert den Konnektor als Universal-Konnektor.

Konstruktoren

\+ KonUniversal ()
Der Konstruktor setzt den Typ dieses Objektes.

Methoden

\+ exportQT (*EasyFileWriter*) → *void*
überschreibt die Methode der Klasse *LogPunkt*.
Exportiert diesen Universalkonnektor in den EasyFileWriter.

D.11.6 Klasse LogComponent

(abstrakte Klasse)
extends *score.util.Observable*
Diese Klasse ist die abstrakte Oberklasse aller Elemente im logischen Statechart Modell. Sie wird vom LogSCModell beobachtet und leitet einen Refresh-Aufruf an dieses weiter.

Attribute

+$ doRefresh:*boolean* – Gibt an, ob ein Refresh wirklich weitergegeben werden soll.

Konstruktoren

\+ LogComponent ()
Erschafft ein neues LogComponent. Der Observer wird entsprechend gesetzt.

Methoden

+$ setObserver (*Observer*) → *void*
Setzt den Observer für alle LogComponents, die zukünftig erschaffen werden.

\+ refresh (*Observable*) → *void*
Leitet ein Refresh an das LogSCModell weiter.

\+ refresh () → *void*
Leitet ein Refresh an das LogSCModell weiter.

D.11.7 Klasse LogPfeil

(abstrakte Klasse)
extends *score.LogSCM.LogComponent*
implements *score.Aktionen.Actor*

Diese Klasse stellt eine logische Verbindung dar.

Attribute

\# event:*Ereignis* – das Ereignis, das diesen Übergang auslöst

\# condition:*Bedingung* – die Bedingung, die diesen Übergang kontrolliert

\# actions:*AktKomplex* – die komplexe Aktion, die bei diesem Übergang ausgeführt wird

\# type:*int* – der Typ dieses Pfeils

+$ ARROW_1_1:*int* CONST – Konstante, falls es sich um einen 1-1 Übergang handelt

+$ ARROW_1_N:*int* CONST – Konstante, falls es sich um einen 1-N Übergang handelt

+$ ARROW_N_1:*int* CONST – Konstante, falls es sich um einen N-1 Übergang handelt

Konstruktoren

\# LogPfeil (*int*)

Setzt den logischen Pfeil auf den folgenden Anfangszustand: Es gibt kein notwendiges Ereignis, die Bedingung ist immer erfüllt, keine Aktion wird ausgeführt.

\+ LogPfeil ()
Erschafft einen leeren LogPfeil für das ausführbare Programm

Methoden

\+ doesAction (*Aktion*) → *boolean*
Implementation des Actor-Interfaces. Gibt zurück, ob dieser Übergang die übergebene Aktion aufruft.

\+ getActorType () → *int*
Implementation des Actor-Interfaces. Gibt zurück, dass es sich um einen Pfeil handelt.

+$ getLogPfeil (*int*) → *LogPfeil*
Gibt den logischen Pfeil mit den angegebenen Identifier zurück.

+$ getAll () → *Collection*
Gibt eine Collection aller LogPfeil zurück.

\+ getID () → *int*
Gibt den Identifier dieses logischen Pfeils zurück.

\+ setEvent (*Ereignis*) → *void*
Setzt das Ereignis, das diesen Pfeil ausl;st.

\+ getEvent () → *Ereignis*
Gibt das Ereignis zurück, das zu diesem Pfeil gehört.

+$ getAllTriggeredBy (*Ereignis*) → *Vector*
Gibt einen Vector von LogPfeilen zurück, die das übergebene Ereignis als Trigger haben. Gibt es keine solchen, oder ist das Ereignis `Null`, so wird ein leerer Vector zurückgegeben.

\+ setCondition (*Bedingung*) → *void*
Setzt die Bedingung, die zu diesem Pfeil gehört.

\+ getCondition () → *Bedingung*
Gibt die Bedingung zurück, die zu diesem Pfeil gehört.

\+ getCalledAction () → *CalledAction*
Implementation des Actor-Interfaces. Gibt die komplexe Aktion zurück, die von diesem Pfeil aufgerufen wird.

\+ getCalledAction (*int*) → *CalledAction*
Implementation des Actor-Interfaces. Gibt die komplexe Aktion zurück, die von diesem Pfeil aufgerufen wird.

\+ removeAction (*Aktion*) → *void*
Entfernt die übergebene Aktion aus diesem Pfeil.

\+ getCompAction () → *AktKomplex*
Gibt die komplexe Aktion zurück.

\+ setCompAction (*AktKomplex*) → *void*
Setzt die komplexe Aktion dieses LogPfeil auf den übergebenen Wert.

\+ calcScope () → *void*
Berechnet den ersten Zustand, der alle Start- und Endzustände enthält.

\+ setScope (*LogPunkt*) → *void*
Setzt den Scope auf den übergebenen LogPunkt.

\+ getScope () → *LogPunkt*
Gibt den ersten Zustand zurück, der alle Start- und Endzustände enthält. Dieser Zustand wird in dieser Methode *nicht* neu berechnet.

\+ redirect (*LogPunkt, LogPunkt, boolean*) → *void*

Ersetzt Start- und Endzustände, die dem ersten übergebenen LogPunkt entsprechen, durch den zweiten. Muss von den einzelnen PfeilTypen entsprechend implementiert werden.

\+ isBasicallyPossible () → *boolean*

Überprüft Ereignis und Bediungung dieses Übergangs.

\+ is1_1 () → *boolean*

Gibt zurück, ob dies ein 1-1 Pfeil ist.

\+ is1_N () → *boolean*

Gibt zurück, ob dies ein 1-N Pfeil ist.

\+ isN_1 () → *boolean*

Gibt zurück, ob dies ein N-1 Pfeil ist.

\+ getType () → *int*

Gibt den Typ dieses Pfeils zurück.

\+ isEmpty () → *boolean*

Gibt zurück, ob dieser Pfeil 'leer' ist (also weder Ereignis, Bedingung noch Aktionen besitzt).

\+ get1_1 () → *LogPfeil*

Gibt einen 1-1 Pfeil zurück. Diese Methode muss von den einzelnen Pfeiltypen entsprechend implementiert werden.

\+ delete () → *void*

überschreibt die Methode der Klasse *Observable*.
Löscht diesen Pfeil. Die Bedingung und komplexe Aktion werden auch gelöscht, das Triggerereignis und die eingetragenenn atomaren Aktionen jedoch nicht.

\# deleteSoft () → *void*

Entfernt diesen Pfeil aus der Liste aller LogPfeil. Start- und Endzustände bleiben unberührt. Die enthaltene Bedingung wird jedoch gelöscht.

\+ saveProps (*EasyFileWriter*) → *void*

Speichert den Befehl, das Trigger-Ereignis und die Bedingung hinzuzufügen.

\+$ exportAll (*EasyFileWriter*) → *void*

Exportiert den Quelltext für alle LogPfeile in den EasyFileWriter.

\+$ exportAllConditions (*EasyFileWriter*) → *void*

Exportiert die Bedingungen aller LogPfeile in den EasyFileWriter. Dabei werden nur die Aufrufe der entsprechenden atomaren Bedingungen geschrieben, nicht der Quelltext dieser Bedingungen selbst.

\+ exportQT (*EasyFileWriter*) → *boolean*

Diese abstrakte Methode exportiert den Quelltext dieses Pfeils, wenn sie entsprechend implementiert wurde.

D.11.8 Klasse LogPfeil_1_1

extends *score.LogSCM.LogPfeil*
Diese Klasse stellt eine logische Verbindung dar. Sie besitzt einen Start- und einen Zielzustand.

Konstruktoren

\+ LogPfeil_1_1 ()

Der Pfeil besitzt am Anfang weder Start- noch Endzustand.

\+ LogPfeil_1_1 (*LogPunkt, LogPunkt*)

Erzeugt einen 1-1 Pfeil mit gegebenem logischen Start- und Endpunkt.

Methoden

\+ setSource (*LogPunkt*) → *void*

Setzt den Startzustand auf den übergebenen LogPunkt.

+ setDest (*LogPunkt*) → *void*

  Setzt den Endzustand auf den übergebenen LogPunkt.

+ getSource () → *LogPunkt*

  Gibt den LogPunkt zurück, bei dem dieser Pfeil beginnt.

+ getDest () → *LogPunkt*

  Gibt den LogPunkt zurück, bei dem dieser Pfeil endet.

+ redirect (*LogPunkt, LogPunkt, boolean*) → *void*

  überschreibt die Methode der Klasse *LogPfeil*.
  Ersetzt Start- und Endpunkt, falls sie dem ersten LogPunkt entsprechen, durch den zweiten. Der Parameter `all` wird ignoriert.

+ get1_1 () → *LogPfeil*

  überschreibt die Methode der Klasse *LogPfeil*.
  Gibt diesen Pfeil zurück, da er bereits ein 1-1 Pfeil ist.

+ delete () → *void*

  überschreibt die Methode der Klasse *LogPfeil*.
  Löscht diesen Pfeil; dabei wird er auch aus Start- und Endzustand entfernt.

+ exportQT (*EasyFileWriter*) → *boolean*

  überschreibt die Methode der Klasse *LogPfeil*.
  Exportiert den Quelltext für diesen LogPfeil in den EasyFileWriter.

D.11.9 Klasse LogPfeil_1_N

extends *score.LogSCM.LogPfeil*
Diese Klasse stellt eine logische Verbindung dar. Sie besitzt einen Start- und einen Zielzustand.

Konstruktoren

+ LogPfeil_1_N ()

  Der Pfeil besitzt am Anfang weder Start- noch Endzustand.

+ LogPfeil_1_N (*LogPunkt, Vector*)

  Der neue Pfeil enthält den gegebenen logischen Startzustand und die Endzustände aus dem Vector.
  Es wird davon ausgegangen, dass der Vector nur logische Punkte enthält.

+ LogPfeil_1_N (*LogPfeil_1_1*)

  Erschafft einen 1-N Pfeil aus einem 1-1 Pfeil.

Methoden

+ getSource () → *LogPunkt*

  Gibt den LogPunkt zurück, von dem dieser Pfeil ausgeht.

+ getDest () → *Vector*

  Gibt einen Vector mit allen LogPunkten zurück, bei denen dieser Pfeil endet.

+ setSource (*LogPunkt*) → *void*

  Setzt den Startpunkt auf den übergebenen LogPunkt.

+ addDest (*LogPunkt*) → *void*

  Fügt den übergebenen LogPunkt zu den Endpunkten hinzu.

+ removeDest (*LogPunkt*) → *void*

  Entfernt den übergebenen LogPunkt aus der Liste der Endzustände.

+ redirect (*LogPunkt, LogPunkt, boolean*) → *void*

überschreibt die Methode der Klasse *LogPfeil*.
Ersetzt alle Start- und Endpunkte, die dem ersten LogPunkt entsprechen, durch den zweiten. Ist der Parameter `all` auf `false` gesetzt, wird nur der erste Endpunkt ersetzt.

+ get1_1 () → *LogPfeil*
  überschreibt die Methode der Klasse *LogPfeil*.
  Wenn nur ein einziger Endzustand vorhanden ist, wird ein 1-1 Pfeil zurückgegeben, der diesem 1-N Pfeil entspricht. Start- und Endzustände werden auf den neuen Pfeil umgesetzt, und dieser alte Pfeil aus der Liste aller LogPfeil entfernt. Sind jedoch mehrere Endzustände vorhanden, wird `Null` zurückgegeben.
+ delete () → *void*
  überschreibt die Methode der Klasse *LogPfeil*.
  Löscht diesen LogPfeil; dabei wird er auch aus allen Start- und Endzuständen entfernt.
+ exportQT (*EasyFileWriter*) → *boolean*
  überschreibt die Methode der Klasse *LogPfeil*.
  Exportiert den Quelltext für diesen LogPfeil in den EasyFileWriter.

D.11.10 Klasse LogPfeil_N_1

extends *score.LogSCM.LogPfeil*
Diese Klasse stellt eine logische Verbindung dar. Sie besitzt einen Start- und einen Zielzustand.

Konstruktoren

+ LogPfeil_N_1 ()
  Der Pfeil besitzt am Anfang weder Start- noch Endzustand.
+ LogPfeil_N_1 (*Vector, LogPunkt*)
  Der Pfeil besitzt die Startzustände des Vectors und als Endzustand den übergebenen logischen Punkt. Es wird davon ausgegangen, dass er Vector ebenfalls logische Punkte enthält.
+ LogPfeil_N_1 (*LogPfeil_1_1*)
  Erschafft einen neuen N-1 Pfeil aus einem 1-1 Pfeil.

Methoden

+ getSource () → *Vector*
  Gibt einen Vector zurück, der sämtliche LogPunkte enthält, von denen dieser Pfeil augeht.
+ getDest () → *LogPunkt*
  Gibt den LogPunkt zurück, bei dem dieser Pfeil endet.
+ addSource (*LogPunkt*) → *void*
  Fügt einen weiteren LogPunkt als Startpunkt hinzu.
+ removeSource (*LogPunkt*) → *void*
  Entfernt den übergebenen LogPunkt aus der Liste der Startzustände.
+ setDest (*LogPunkt*) → *void*
  Setzt den Endpunkt dieses Pfeils auf den angegebenen LogPunkt.
+ redirect (*LogPunkt, LogPunkt, boolean*) → *void*
  überschreibt die Methode der Klasse *LogPfeil*.
  Ersetzt alle Start- und Endzustände, die dem ersten LogPunkt entsprechen, durch den zweiten. Ist der Parameter `all` auf `false` gesetzt, wird nur der erste Startpunkt ersetzt.
+ get1_1 () → *LogPfeil*
  überschreibt die Methode der Klasse *LogPfeil*.
  Wenn nur ein einziger Startzustand vorhanden ist, wird wird ein 1-1 Pfeil zurückgegeben, der diesem 1-N Pfeil entspricht. Start- und Endzustände werden auf den neuen Pfeil umgesetzt, und dieser alte Pfeil aus der Liste aller LogPfeil entfernt. Sind jedoch mehrere Startzustände vorhanden, wird `Null` zurückgegeben.
+ delete () → *void*

überschreibt die Methode der Klasse *LogPfeil*.
Löscht diesen LogPfeil; er wird dabei auch aus allen Start- und Endzuständen entfernt.

\+ exportQT (*EasyFileWriter*) → *boolean*
überschreibt die Methode der Klasse *LogPfeil*.
Exportiert den Quelltext für diesen LogPfeil in den EasyFileWriter.

D.11.11 Klasse LogPunkt

(abstrakte Klasse)
extends *score.LogSCM.LogComponent*
Diese abstrakte Klasse stellt einen Punkt im logischen Statechart dar. Ein Punkt kann als Start- oder Endpunkt von Verbindungen benutzt werden.

Attribute

\# type:*int* – der Typ dieser Instanz

\#$ TYP_CONN_UNIV:*int* CONST – Konstante für einen Universal-Konnektor

\#$ TYP_CONN_HIST_FLAT:*int* CONST – Konstante für einen History-Konnektor

\#$ TYP_CONN_HIST_DEEP:*int* CONST – Konstante für einen Deep-History-Konnektor

\#$ TYP_CONN_DEF:*int* CONST – Konstante für Default-Einstiegs-Konnektoren

\#$ TYP_ST_ATOM:*int* CONST – Konstante für einen atomaren Zustand

\#$ TYP_ST_XOR:*int* CONST – Konstante für einen OR-Zustand

\#$ TYP_ST_AND:*int* CONST – Konstante für einen AND-Zustand

Konstruktoren

\# LogPunkt ()
Der Konstruktor inizialisiert die Sammlungen der eingehenden und ausgehenden Verbindungen, von denen anfangs noch keine vorhanden sind. Der Typ dieser abstrakten Klasse wird auf einen ungültigen Wert gesetzt. Der Identifier wird mit dem nächsten Wert initialisiert.

\# LogPunkt (*LogPunkt*)
Dieser Konstruktor ersetzt einen logischen Punkt in jeglicher Hinsicht durch einen neuen. Der einzige Unterschied zwischen den beiden Punkten ist der Typ, der bei dem neuen Zustand noch auf einen ungültigen Wert gesetzt ist. Der neue Punkt wird mit dem alten Identifier in allLogPunkt eingefügt.

\# LogPunkt (*int, LogPunkt*)
Erschafft einen LogPunkt als Oberklasse des SimBasic in der Simulation. Der LogPunkt wird in diesem Fall nicht zum Hashtable der LogPunkte hinzugefügt.

Methoden

\+$ getLogPunkt (*int*) → *LogPunkt*
Gibt den logischen Punkt mit der angegebenen Nummer zurück. Wenn kein solcher Punkt existiert, wird `Null` zurückgegeben.

\+$ getAll () → *Collection*
Gibt eine Collection mit allen logischen Punkten zurück.

\+$ getAllStatesWithHistory (*boolean*) → *Vector*
Liefert Vector aller Zustände mit History bzw. DeepHistory

\+$ swapIdents (*LogPunkt, LogPunkt*) → *void*
Vertauscht Identifier von zwei logischen Punkten.

\+ getID () → *int*
Gibt den Identifier dieses logischen Punktes zurück.

\+ getIdent () → *String*

Gibt eine eindeutige, lesbare Bezeichnung dieses logischen Punktes zurück.

+ getIncoming () → *Vector*

  Gibt die eingehenden Verbindungen dieses logischen Punktes in einem Vektor zurück. Gibt es keine, wird zurückgegeben.

+ getOutgoing () → *Vector*

  Gibt die ausgehenden Verbindungen dieses logischen Punktes in einem Vektor zurück. Gibt es keine, wird ein leerer Vector zurückgegeben.

+ setSuper (*LogPunkt*) → *boolean*

  Setzt den übergeordneten Zustand dieses logischen Punktes. Ist der angegebene logische Punkt kein zusammengesetzter Zustand oder , wird `false` zurückgegeben, aber der Wert dennoch gesetzt.

+ getSuper () → *LogPunkt*

  Gibt den als übergeordneten Zustand angegebenen logischen Punkt zurück. Es wird nicht garantiert, dass dieser Wert nicht ist.

+ isConnector () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Konnektor handelt. Hier wird nicht zwischen Universal-Konnektor und History-Konnektor unterschieden.

+ isUniversal () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Universalkonnektor handelt.

+ isHistory () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Histroykonnektor handelt.

+ isFlatHistory () → *boolean*

  Gibt `true` zurück, wenn es sich um einen flachen Historykonnektor handelt.

+ isDeepHistory () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Deep-History-Konnektor handelt.

+ isDefEntry () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Default-Entry-Konnektor handelt.

+ isState () → *boolean*

  Gibt `true` zurück, wenn es sich um einen Zustand handelt. Hier wird nicht zwischen atomaren, OR- oder AND-Zuständen unterschieden.

+ isBasic () → *boolean*

  Gibt `true` zurück, wenn es sich um einen atomaren Zustand handelt.

+ isComplex () → *boolean*

  Gibt `true` zurück, wenn es sich um einen komplexen Zustand handelt.

+ isOR () → *boolean*

  Gibt `true` zurück, wenn es sich um einen OR-Zustand handelt.

+ isAND () → *boolean*

  Gibt `true` zurück, wenn es sich um einen AND-Zustand handelt.

+ addIncoming (*LogPfeil*) → *void*

  Fügt eine Verbindung zu den eingehenden Verbindungen dieses logischen Punktes hinzu. Es wird nicht überprüft, ob diese Verbindung schon eingetragen war, sie ist aber danach mindestens einmal eingetragen.

+ addOutgoing (*LogPfeil*) → *void*

  Fügt eine Verbindung zu den ausgehenden Verbindungen dieses logischen Punktes hinzu. Es wird nicht überprüft, ob diese Verbindung schon eingetragen war, sie ist aber danach mindestens einmal eingetragen.

+ removeIncoming (*LogPfeil*) → *void*

Löscht eine Verbindung aus den eingehenden Verbindungen, auch wenn sie mehrfach eingetragen war.

+ removeOutgoing (*LogPfeil*) → *void*

  Löscht eine Verbindung aus den ausgehenden Verbindungen, auch wenn sie mehrfach eingetragen war.

+ substatesExist () → *boolean*

  Gibt an, ob dieser logische Punkt Unterpunkte hat.

+ translate (*Hashtable[]*) → *LogPunkt*

  Gibt eine Übersetzung dieses LogPunkts zurück.

+ exportQT (*EasyFileWriter*) → *void*

  Exportiert den Quelltext für diesen LogPunkt in den EasyFileWriter.

D.11.12 Klasse LogSCModell

extends *score.util.Observable*
implements *score.util.Observer*

Diese Klasse verwaltet das komplette logische Statechart Modell. Sie implementiert das Interface Askable für die Ereignisse und Bedingungen.

Konstruktoren

+ LogSCModell ()

  Erschafft ein logisches Statechart Modell. Dieses wird bei den Ereignissen und Bedingungen gleich als Beantworter der Wahrheitswerte eingetragen. Außerdem wird ein neuer atomarer Zustand erzeugt, der als globaler Wurzelzustand eingetragen wird.

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*

  Implementation des Observable-Interfaces.

+ getTopLevelState () → *LogPunkt*

  Gibt den globalen Wurzelzustand zurück.

+ getLogPunkt (*int*) → *LogPunkt*

  Gibt den logischen Punkt mit den angegebenen Identifizierer zurück. Wenn es keinen solchen Punkt gibt, wird `Null` zurückgegeben.

+ update (*Observable, Object*) → *void*

  Implementation des Observer-Interfaces. Gibt Aufrufe an die Observer des LogSCModells weiter.

+ exportQT (*EasyFileWriter*) → *void*

  Exportiert das gesamte logische Modell in den EasyFileWriter. Dabei wird die Methode generateSCModell erzeugt.

+ exportConditions (*EasyFileWriter*) → *void*

  Exportiert sämtliche Bedingungen in den EasyFileWriter. Hierbei wird die Klasse *ExportBed* komplett geschrieben.

+ exportActions (*EasyFileWriter*) → *void*

  Exportiert sämtliche Aktionen in den EasyFileWriter. Hierbei wird die Klasse *ExportAkt* komplett geschrieben.

D.11.13 Klasse ZustAND

extends *score.LogSCM.ZustKomplex*
Diese Klasse stellt einen AND-verfeinerten Zustand dar.

Attribute

– **$** newlyActivatedANDs:*Vector* – In dieser Klassenvariable werden die AND-Zustände gesammelt, die gerade aktiviert wurden. Um eine Aktivierung abzuschließen, müssen diese Zustände nachträglich aktiviert werden, um sicherzustellen, dass alle Partitionen aktiviert sind.

Konstruktoren

+ ZustAND ()

Der Konstruktor setzt den Typ dieser Instanz.

+ ZustAND (*LogPunkt*)

Erschafft einen neuen ZustAND, der einen alten logischen Punkt ersetzt.

Methoden

+ addPartition (*ZustOR*) → *void*

Fügt einen ZustOR als Partition dieses Zustandes hinzu. Für diesen wird der Oberzustand gesetzt.

+ removePartition (*ZustOR*) → *void*

Entfernt einen ZustOR, der eine Partition dieses Zustands ist. Für diesen wird der Oberzustand auf `Null` gesetzt.

+ exportQT (*EasyFileWriter*) → *void*

überschreibt die Methode der Klasse *ZustandAtom*.
Exportiert diesen AND-Zustand in den EasyFileWriter.

D.11.14 Klasse ZustandAtom

extends *score.LogSCM.LogPunkt*
implements *score.Aktionen.Actor*

Diese Klasse erweitertn den logischen Punkt zu einem Atomaren Zustand.

Attribute

+**$** createDefaultActions:*boolean* – Gibt an, ob beim Erschaffen eines Zustands die Aktionen `throw!(enter.())` und `throw!(exit.())` erschaffen und eingefügt werden sollen.

Konstruktoren

+ ZustandAtom ()

Der Konstruktor setzt diesen atomaren Zustand auf den folgenden definierten Startzustand: Für das Aktivieren und Deaktivieren sind keine Ereignisse gesetzt. Der Zustand ist deaktiviert.

+ ZustandAtom (*LogPunkt*)

Erzeugt einen Zustand aus einem logischen Punkt; der neue Zustand ersetzt den alten Punkt.

Methoden

+ delete () → *void*

überschreibt die Methode der Klasse *Observable*.
Löscht diesen Zustand und die Aktionen, die nur von ihm ausgelöst werden.

+**$** swapActions (*ZustandAtom, ZustandAtom*) → *void*

Tauscht die Aktionen der beiden Zustände aus.

+ getActorType () → *int*

Implementation des Actor-Interfaces; gibt zurück, dass es sich um einen Zustand handelt.

+ doesAction (*Aktion*) → *boolean*

  Implementation des Actor-Interfaces; gibt zurück, ob dieser Zustand die übergebene Aktion beim Aktivieren oder Deaktivieren auslöst.

+ removeAction (*Aktion*) → *void*

  Entfernt die übergebene Aktion aus diesem Zustand; sowohl aus den Aktionen beim Aktivieren als auch beim Deaktivieren.

+ getCalledAction (*int*) → *CalledAction*

  Implementation des Actor-Interfaces; gibt die komplexe Aktion zurück, die angefragt wurde. Übergeben wird dabei Actor.ACTIVATE, falls es sich um die Aktion beim Aktivieren handelt soll, und Actor.DEACTIVATE, falls es sich um die Aktion beim Deaktivieren handeln soll.

+ getCalledAction () → *CalledAction*

  Gibt `Null` zurück, da diese Anfrage bei einem Zustand nicht genau genug spezifiziert ist.

+ getKompAction (*int*) → *AktKomplex*

  Gibt die komplexe Aktion zurück.

+ isTerminal () → *boolean*

  Gibt an, ob der Zustand terminal ist, d. h. ob er ausgehende Verbindungen besitzt. Verbindungen, die von Oberzuständen ausgehen, werden dabei nicht beachtet.

+ isEndState () → *boolean*

  Gibt an, ob der Zustand als Endzustand markiert ist.

+ setEndState (*boolean*) → *void*

  Diese Methode setzt den Wert, der angibt, ob es sich bei diesem Zustand um einen Endzustand handelt. Wird dadurch der Wert geändert, wird ein Refresh aufgerufen.

+ exportQT (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *LogPunkt*.
  Exportiert den Quelltext für diesen atomaren Zustand in den EasyFileWriter.

exportAtom (*EasyFileWriter*) → *void*

  Schreibt Identifier, Oberzustand und Aktionen dieses Zustands in den EasyFileWriter.

D.11.15 Klasse ZustKomplex

(abstrakte Klasse)
extends *score.LogSCM.ZustandAtom*
Diese abstrakte Klasse liegt zwischen einem atomaren und den konkreten zusammengesetzten Zuständen.

Attribute

subStates:*Vector* – Hält die direkten Unterzustände dieses Zustands; bei AND-Zuständen sind dies die Partitionen.

Konstruktoren

ZustKomplex ()

  Der Konstruktor erzeugt einen neuen (leeren) Vektor von Unterzuständen.

ZustKomplex (*LogPunkt*)

  Erschafft einen neuen ZustKomplex, der einen alten logischen Punkt ersetzt.

Methoden

+ removeSubStates () → *void*

  Löscht die Eintragung sämtlicher Unterzustände.

replaceSubState (*LogPunkt, LogPunkt*) → *void*

überschreibt die Methode der Klasse *LogPunkt*.
Ersetzt einen Unterzustand durch einen anderen. Wird der erste logische Punkt nicht als Unterzustand gefunden, geschieht nichts.

+ substatesExist () → *boolean*

  überschreibt die Methode der Klasse *LogPunkt*.
  Gibt `true` zurück, wenn mindestens ein Unterzustand existiert.

+ getSubStates () → *Vector*

  Gibt einen Vector mit den Unterzuständen zurück.

exportSubstates (*EasyFileWriter*) → *void*

  Exportiert die Unterzustände in des EasyFileWriter.

D.11.16 Klasse ZustOR

extends *score.LogSCM.ZustKomplex*
Diese Klasse stellt einen OR-verfeinerten Zustand dar.

Konstruktoren

+ ZustOR ()

  Der Konstruktor erzeugt einen OR-Zustand; die History ist leer und es existiert kein Default-Einstieg.

+ ZustOR (*LogPunkt*)

  Erschafft einen neuen ZustOR, der einen alten logischen Punkt ersetzt.

Methoden

+ addSubPoint (*LogPunkt*) → *void*

  Fügt einen logischen Punkt als direkten Unterpunkt dieses Zustandes hinzu. Für diesen wird der Oberzustand gesetzt.

+ removeSubPoint (*LogPunkt*) → *void*

  Entfernt einen Unterzustand. Falls dieser in der Liste der direkten Unterzustände dieses Zustands vorhanden war, wird der Oberzustand des entferntes Zustands auf `Null` gesetzt.

+ setDefault (*LogPunkt*) → *void*

  Setzt den Default-Einstieg. Es wird nicht geprüft, ob der angegebene logische Punkt wirklich ein direkter Unterzustand (oder History-Konnektor) dieses Zustands ist; dies wird aber beim Benutzen vorausgesetzt. Die Angabe löscht den Default-Einstieg.

+ setDefault (*int*) → *void*

  Setzt den Default-Einstieg auf den Zustand mit dem übergebenen Identifier.

+ getDefEntry () → *LogPunkt*

  Gibt den Unterzustand zurück, der als Default-Einstieg markiert ist.

+ calcDefEntry () → *boolean*

  Berechnet den Default-Einstieg aus dem Grafischen Modell. Es wird davon ausgegangen, dass die Konstruktion gültig ist; ansonsten wird `false` zurückgegeben.

+ exportQT (*EasyFileWriter*) → *void*

  überschreibt die Methode der Klasse *ZustandAtom*.
  Schreibt den Quelltext zu diesem OR-Zustand in den EasyFileWriter.

D.12 Quelltext-Export

D.12.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *javax.swing.JButton*

↪ *Timer*
↪ *score.Simulator.SCRun*
↪ *StateChart*
↪ *score.Simulator.Variable*
↪ *ExportVariable*

D.12.2 Klasse ExportVariable

extends *score.Simulator.Variable*
Diese Klasse erweiter die Variable für das exportierte Programm

Konstruktoren

+ ExportVariable (*String*)
  Konstruktor
+ ExportVariable (*String, String*)
  Konstruktor
+ ExportVariable (*String, int*)
  Konstruktor
+ ExportVariable (*String, double*)
  Konstruktor

Methoden

checkDialogVar () → *void*
  überschreibt die Methode der Klasse *Variable*.
  Prüft ob diese Variable zu einer DialogComponenten gehört

D.12.3 Klasse StateChart

(abstrakte Klasse)
extends *score.Simulator.SCRun*
implements *score.LogDM.DialogCompServer,*
score.LogDM.DialogCompManager,
java.awt.event.ActionListener,
java.awt.event.WindowListener,
score.Ereignisse.EventAskable,
score.control.Logger

Diese Klasse führt den Ablauf des Statechart im exportierten Programm durch. Sie erweitert dafür SCRun und implementiert die DialogComponentenzugriff-Interfaces

Attribute

dlg:*Hashtable* – Hashtable in der die DialogKomponenten gehalten werden. Der Key ist hierbei die JComponent für den Zugriff mit Visible und der Value die JComponent für den Zugriff mit Variablen

Konstruktoren

+ StateChart ()
  Konstruktor

Methoden

+ getComponentWithId (*int*) → *Object*
  Implementation des DialogCompServer Interfaces
setPhase (*int*) → *void*

überschreibt die Methode der Klasse *SCRun*.
Setzt die Phase auf phase

+ setDlgVisible (*int, boolean*) → *void*

  Implementation des DialogCompServer Interfaces

+ setOn (*int*) → *void*

  Implementation des DialogCompServer Interfaces

+ setOff (*int*) → *void*

  Implementation des DialogCompServer Interfaces

+ occurred (*Ereignis*) → *boolean*

  Implementation des EventAskable Interfaces

createDialogs () → *void*

  überschreibt die Methode der Klasse *SCRun*.
  Erzeugt die Dialoge

fetchDlgActions () → *void*

  überschreibt die Methode der Klasse *SCRun*.
  Holt die Aktionen aus den Dialogkomponenten

+ throwEvent (*String*) → *void*

  Speichert ein Event

+ getVariable (*String*) → *Variable*

  Für den VarServer

+ addLine (*String*) → *void*

  Für den Logger

checkForEndStates () → *void*

  überschreibt die Methode der Klasse *SCRun*.
  Prüft ob ein EndState erreicht wurde und ruft dann endStateChart() auf

D.12.4 Klasse Timer

extends *javax.swing.JButton*
implements *java.lang.Runnable*

Diese Klasse stellt einen Timer im Export dar. Dabei wird der Timer wie ein JButton benutzt.

Konstruktoren

+ Timer (*int*)

  Konstruktor mit Startwert

Methoden

+ setValue (*int*) → *void*

  Den Timer Wert neu setzen

+ setSelected (*boolean*) → *void*

  überschreibt die Methode der Klasse *AbstractButton*.
  Starten und stoppen des Timers über die Zugriffsfunktion von AbstractButton

+ run () → *void*

  Runnable-Interface: Zählt hinunter und macht ein doClick() bei 0

D.13 Package score.SCEditor

D.13.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *java.awt.Canvas*
    ↪ *score.SCEditor.BufferedCanvas*
  ↪ *java.awt.ScrollPane*
    ↪ *score.SCEditor.ScrollCanvas*
      ↪ *score.SCEditor.SCCanvas*
  ↪ *java.awt.event.WindowAdapter*
    ↪ *score.SCEditor.SCEditor*
  ↪ *score.SCEditor.DrawableRubber*
    ↪ *score.SCEditor.RubberBand*
    ↪ *score.SCEditor.RubberRectangle*
      ↪ *score.SCEditor.RubberOval*
        ↪ *score.SCEditor.RubberCircle*
          ↪ *score.SCEditor.RubberFilledCircle*
      ↪ *score.SCEditor.RubberRoundRect*
    ↪ *score.SCEditor.RubberSet*
    ↪ *score.SCEditor.RubberText*
  ↪ *score.SCEditor.Painter*
  ↪ *score.util.Observable*
    ↪ *score.SCEditor.TranslationWindow*

D.13.2 Klasse BufferedCanvas

extends *java.awt.Canvas*
Erweitert das Canvas um DoubleBuffering

Konstruktoren

+ BufferedCanvas (*int, int, ScrollPane*)
  Konstruktor
+ BufferedCanvas (*int, int*)
  Konstruktor

Methoden

+ setSize (*Dimension*) → *void*
  überschreibt die Methode der Klasse *Component*.
  Setzt die Größe neu
+ setSize (*int, int*) → *void*
  überschreibt die Methode der Klasse *Component*.
  Setzt die Größe neu
+ getGraphics () → *Graphics*
  überschreibt die Methode der Klasse *Component*.
  Liefert Graphics des Buffers
+ getUnbufferedGraphics () → *Graphics*
  Liefert Graphics der Zeichenfläche direkt
+ paint (*Graphics*) → *void*
  überschreibt die Methode der Klasse *Canvas*.
  Zeichnen des Buffers in die Zeichenfläche
paint (*Point, Dimension*) → *void*
  Zeichnen von Position p in Größe d

D.13.3 Klasse DrawableRubber

(abstrakte Klasse)
extends *java.lang.Object*
Abstrakte Oberklasse für Feedback-Objekte (RubberBand etc.) für Mauseingaben

Attribute

\# g:*Graphics* – Referenz auf das zu zeichnende Graphics

\# a:*Point* – Obere linke, untere Rechte Ecke und ein Offset

\# b:*Point* – Obere linke, untere Rechte Ecke und ein Offset

\# offset:*Point* – Obere linke, untere Rechte Ecke und ein Offset

\# c:*Color* – Farbe

\# visible:*boolean* – Ist dieser Rubber momentan sichtbar?

Methoden

\+ draw () → *void*
Rubber zeichnen

\+ erase () → *void*
Rubber löschen

\+ getPos () → *Point*
Position auslesen

\+ getA () → *Point*
Liefer Obere linke Ecke

\+ getB () → *Point*
Liefer untere rechte Ecke

\+ setA (*Point*) → *void*
Setzt obere linke Ecke

\+ setB (*Point*) → *void*
Setzt untere rechte Ecke

\+ getOffset () → *Point*
Liefert den Zeichenoffset

\+ setOffset (*Point*) → *void*
Setzt den Zeichenoffset

\+ getSize () → *Dimension*
Liefert die Größe

\+ moveTo (*Point*) → *void*
Bewegt an Position p

\+ moveToX (*int*) → *void*
Bewegt in X-Richtung

\+ moveToY (*int*) → *void*
Bewegt in Y-Richtung

\+ setSize (*Dimension*) → *void*
Setzt die Größe neu

D.13.4 Klasse Painter

extends *java.lang.Object*
Stellt die Hilevel Zeichenroutinen zur Verfügung

Attribute

+$ sim:*Simulation* – Referenz auf die Simulation

+$ COLOR_DEFAULT:*Color* CONST – Vordefinierte Farbkonstanten

+$ COLOR_COMPLEX:*Color* CONST – Vordefinierte Farbkonstanten

+$ COLOR_ACTIVE:*Color* CONST – Vordefinierte Farbkonstanten

+$ COLOR_MARKED:*Color* CONST – Vordefinierte Farbkonstanten

Konstruktoren

+ Painter (*ZoomGraphics, GrafModellSicht*)
Konstruktor

Methoden

+ drawState (*Point2D.Double, GrafZustand*) → *void*
Einen Zustand zeichnen

+ setColorToDefault () → *void*
Default-Farbe setzen

+ drawText (*Point2D.Double, GrafText, GrafModellSicht*) → *void*
Textlabel zeichen

+ drawPartition (*Point2D.Double, GrafZustand*) → *void*
Zeichnet einen Zustand als Partition, d. h. ohne Umrandung.

+ drawPartitioner (*double, double, double, double, Partitioner*) → *void*
Einen Partitioner zeichnen

+ drawArrowSect (*GPAbschnitt, GrafModellSicht, boolean, boolean*) → *void*
Einen Pfeilabschnitt zeichnen

+ drawUniversal (*Point2D.Double, GrafKonnektor*) → *void*
Einen UniversalKonnektor zeichnen

+ drawHistory (*Point2D.Double, GrafKonnektor, boolean*) → *void*
Einen Historyeinstieg zeichnen

+ drawDefault (*Point2D.Double, GrafKonnektor*) → *void*
Einen Defaulteinstieg zeichnen

D.13.5 Klasse RubberBand

extends *score.SCEditor.DrawableRubber*
Ein Gummiband

Attribute

visible:*boolean* – Ist er gerade sichtbar?

Konstruktoren

+ RubberBand (*Graphics, Point, Point, Color*)
Konstruktor

+ RubberBand (*Graphics, Point, Point, Color, Point*)

Konstruktor

Methoden

+ draw () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Rubber zeichnen
+ erase () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Rubber löschen

D.13.6 Klasse RubberCircle

extends *score.SCEditor.RubberOval*
Ein Kreis

Konstruktoren

+ RubberCircle (*Graphics, Point, int, Color*)
  Konstruktor
+ RubberCircle (*Graphics, Point, int, Color, Point*)
  Konstruktor

Methoden

+ setB (*Point*) → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Die zweite Koordinate setzen

D.13.7 Klasse RubberFilledCircle

extends *score.SCEditor.RubberCircle*
Ein gefüllter Kreis

Konstruktoren

+ RubberFilledCircle (*Graphics, Point, int, Color*)
  Konstruktor
+ RubberFilledCircle (*Graphics, Point, int, Color, Point*)
  Konstruktor

Methoden

+ draw () → *void*
  überschreibt die Methode der Klasse *RubberOval*.
  zeichnen

D.13.8 Klasse RubberOval

extends *score.SCEditor.RubberRectangle*
Ein Oval

Konstruktoren

+ RubberOval (*Graphics, Point, Point, Color*)
  Konstruktor
+ RubberOval (*Graphics, Point, Point, Color, Point*)

Konstruktor

+ RubberOval (*Graphics, Point, Dimension, Color, Point*)
  Konstruktor
+ RubberOval (*Graphics, Point, Dimension, Color*)
  Konstruktor

Methoden

+ draw () → *void*
  überschreibt die Methode der Klasse *RubberRectangle*.
  Zeichnen

D.13.9 Klasse RubberRectangle

extends *score.SCEditor.DrawableRubber*
Ein Rechteck

Konstruktoren

+ RubberRectangle (*Graphics, Point, Point, Color*)
  Konstruktor
+ RubberRectangle (*Graphics, Point, Point, Color, Point*)
  Konstruktor
+ RubberRectangle (*Graphics, Point, Dimension, Color, Point*)
  Konstruktor
+ RubberRectangle (*Graphics, Point, Dimension, Color*)
  Konstruktor

Methoden

+ draw () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Zeichnen
+ erase () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Löschen

D.13.10 Klasse RubberRoundRect

extends *score.SCEditor.RubberRectangle*
Ein Rechteck mit runden Ecken

Konstruktoren

+ RubberRoundRect (*Graphics, Point, Point, Color*)
  Konstruktor
+ RubberRoundRect (*Graphics, Point, Point, Color, Point*)
  Konstruktor
+ RubberRoundRect (*Graphics, Point, Dimension, Color*)
  Konstruktor
+ RubberRoundRect (*Graphics, Point, Dimension, Color, Point*)
  Konstruktor

Methoden

+ draw () → *void*
  überschreibt die Methode der Klasse *RubberRectangle*.
  Zeichnen

D.13.11 Klasse RubberSet

extends *score.SCEditor.DrawableRubber*
Diese Klasse merkt sich eine Menge von Rubber-Objekten und bearbeitet sie wie eines

Konstruktoren

+ RubberSet ()
  Konstruktor
+ RubberSet (*Graphics, Point, Enumeration, Color, double, int*)
  Konstruktor
  Parameter:
  - g: Graphics in die die Rubber gezeichnet werden
  - p: Starposition der Bänder
  - e: Aufzählung von Point2D.Double
  - c: Farbe der Bänder
  - zoom: Skalierung
  - rubber_type: Art der Rubber
+ RubberSet (*Graphics, Point, Enumeration, Color, int*)
  Konstruktor

Methoden

+ add (*DrawableRubber*) → *DrawableRubber*
  Einen neuen Rubber zum Set hinzufügen
+ draw () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Alle Rubber zeichnen
+ erase () → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Alle Rubber löschen
+ getB () → *Point*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Liefert auch A
+ setA (*Point*) → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Setzt alle a
+ setB (*Point*) → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Setzt alle b
+ setB (*Point, int*) → *void*
  Setzt b des Elements index
+ setOffset (*Point*) → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Setzt alle Offsets
+ moveTo (*Point*) → *void*
  überschreibt die Methode der Klasse *DrawableRubber*.
  Bewegt alle an Position p
+ moveToX (*int*) → *void*

überschreibt die Methode der Klasse *DrawableRubber*.
Bewegt alle in X

+ moveToY (*int*) → *void*
überschreibt die Methode der Klasse *DrawableRubber*.
Bewegt alle in Y

+ setSize () → *void*
Wird ignoriert

D.13.12 Klasse RubberText

extends *score.SCEditor.DrawableRubber*
Diese Klasse erweitert Rubber für ein Textlabel

Attribute

\# text:*String* – Der zu schreibende Text

Konstruktoren

+ RubberText (*Graphics, Point, Color, String*)
Konstruktor

+ RubberText (*Graphics, Point, Color, Point, String*)
Konstruktor

Methoden

+ draw () → *void*
überschreibt die Methode der Klasse *DrawableRubber*.
Den Text zeichnen

+ erase () → *void*
überschreibt die Methode der Klasse *DrawableRubber*.
Den Text löschen

D.13.13 Klasse SCCanvas

extends *score.SCEditor.ScrollCanvas*
implements *javax.swing.event.MouseInputListener,*
java.awt.event.ActionListener,
score.util.Observer,
java.awt.event.WindowListener

Stellt die Zeichenfläche für das Statechart zur Verfügung

Attribute

\+ gscv:*GrafModellSicht* – Referenz auf die Sicht dieser Darstellung

+$ DEFAULT_SIZE:*Dimension* – Defaultgröße der Canvas bei Start

– editor:*SCEditor* – Parent Editor

+$ CORNER_EPSILON:*int* – HotSpot-Unschärfe für die Ecke

+$ MOUSE_EPSILON:*int* – HotSpot-Unschärfe für die Maus

Konstruktoren

+ SCCanvas (*SCEditor, GrafModellSicht*)
Konstruktor

Methoden

+ replaceObservable (*Observable, Observable*) → *boolean*

Wenn das Modell ausgetauscht wird, disposed der Editor sich

+ repaint () → *void*

überschreibt die Methode der Klasse *Component*.
Das Canvas neu zeichnen

+ print (*String*) → *void*

Das Canvas drucken

+ actionPerformed (*ActionEvent*) → *void*

Auf PopupMenüEvents reagieren

+ mouseClicked (*MouseEvent*) → *void*

Auf einen Mausklick reagieren

+ mousePressed (*MouseEvent*) → *void*

Auf das drücken der Maus reagieren

+ mouseReleased (*MouseEvent*) → *void*

Auf das Loslassen der Maus reagieren

+ mouseMoved (*MouseEvent*) → *void*

Auf Mausbewegungen reagieren

+ mouseDragged (*MouseEvent*) → *void*

Auf das Schieben mit der Maus reagieren

+ setNewView (*GrafModellSicht*) → *void*

Die Sicht neu setzen

+ update (*Observable, Object*) → *void*

Auf eine Modelländerung reagieren -¿ neuzeichnen

D.13.14 Klasse SCEditor

extends *java.awt.event.WindowAdapter*
implements *score.util.ErrorPrinter*

Der Editor für das grafische Statechartmodell

Attribute

- – viewNumber:*int* – Laufende Nummer dieser Ansicht
- – nfDrawDepth:*NumField* – Feld in dem die Darstellungstiefe verwaltet wird
- – nfZoom:*NumField* – Feld in dem die Darstellungstiefe verwaltet wird
- \+ father:*MainWindow* – Rückwärtsreferenz auf das Hauptfenster
- \+ editFrame:*JFrame* – Frame in dem der Editor dargestellt wird
- – gscv:*GrafModellSicht* – Grafische Modellsicht
- – canvas:*SCCanvas* – Die Scrollebene
- – cbGrid:*JCheckBox* – CheckBox für das Gitter
- – cbHooks:*JCheckBox* – CheckBox für das Gitter
- – cbIDs:*JCheckBox* – CheckBox für das Gitter
- – cbLabel:*JCheckBox* – CheckBox für das Gitter
- –$ TOOLCOUNT:*int* CONST – Konstante für die Anzahl möglicher Werkzeuge
- +$ TOOL_CURSOR:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_STATE:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_VPARTITIONER:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_HPARTITIONER:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_ARROW:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_REFLEXIV:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_DEFAULT_CONNECTOR:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_CONNECTOR:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_HIST_CONNECTOR:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_HISTDEEP_CONNECTOR:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_TEXT:*int* CONST – Symbolische Konstante für das Werkzeug
- +$ TOOL_COPY:*int* CONST – Symbolische Konstante für das Werkzeug
- –$ selectedTool:*int* – zeigt an, welches Werkzeug ausgewählt wurde
- – toolListener:*ChangeListener* – ChangeListener, der für alle Werkzeugbuttons eingetragen wird

Konstruktoren

- \+ SCEditor (*GrafModellSicht, int, MainWindow*)
  Konstruktor

Methoden

- – updateToolState () → *void*
  Beim Betreten eines neuen Editorfensters wird mit dieser Funktion das Buttonfeld aktiviert
- – setToolState () → *void*
  Wenn sich der Status der Werkzeugleiste ändert, wird hiermit die interne Variable selectedTool aktualisiert
- \+ getTool () → *int*
  Rückgabewert: Nummer des aktivierten Werkzeugs
  Liefert die Nummer des aktivierten Werkzeugs
- \+ deactivateToolbar () → *void*
  Deaktiviert die Toolbar und setzt auf Werkzeug 0 (Pfeil)
- \+ activateToolbar () → *void*

Aktiviert die Toolbar

+ setZoomLabelSilent (*int*) → *void*
Setzt den ZoomLabel ohne einen refresh auszulösen
Parameter:
  – v: Neuer Zoomwert

+ setZoomLabel (*int*) → *void*
Setzt den ZoomLabel und löst einen refresh auf
Parameter:
  – v: Neuer Zoomwert

+ toFront () → *void*
Holt das Fenster nach vorne

+ print () → *void*
Canvas drucken

– updatePanePositionLabel () → *void*
Aktualisiert den Label mit der relativen Position des Feldes

– init () → *void*
Initialisiert die Komponenten des Editors

– initPane () → *JPanel*
Erstellt die Panel und das grafische Layout

+ getViewName () → *String*
Liefert die Nummer dieser Ansicht

+ setNewModell (*GrafSCModell*) → *void*
Ein neues grafisches Modell für diesen Editor setzen

+ windowActivated (*WindowEvent*) → *void*
überschreibt die Methode der Klasse *WindowAdapter*.
Überschreibt den WindowAdapter Aktualisiert die ToolLeiste

+ windowClosing (*WindowEvent*) → *void*
überschreibt die Methode der Klasse *WindowAdapter*.
Überschreibt den WindowAdapter dipose()

+ dispose () → *void*
Löscht den Eintrag im MainWindow

+ toolCommand (*Point, Point*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*GrafTyp, GrafTyp, Point2D.Double, Point2D.Double*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*Partitioner*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*GrafPunkt*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*Point, GrafZustand*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*Point, GrafTyp*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*Dimension.Double, GrafPunkt*) → *void*
Führt ein Werkzeug-Kommando aus

+ toolCommand (*Point, boolean*) → *void*
Führt ein Werkzeug-Kommando aus

+ printError (*String*) → *void*
Eine Fehlermeldung ausgeben

D.13.15 Klasse ScrollCanvas

extends *java.awt.ScrollPane*
Erweitert einen Canvas um eine ScrollPane

Konstruktoren

+ ScrollCanvas (*int*, *int*)
  Konstruktor

Methoden

+ getCanvas () → *BufferedCanvas*
  Das Canvas liefern
+ getGraphics () → *Graphics*
  überschreibt die Methode der Klasse *Component*.
  Graphics des Canvas liefern
+ getUnbufferedGraphics () → *Graphics*
  Ungepufferte Graphics des Canvas liefern

D.13.16 Klasse TranslationWindow

extends *score.util.Observable*
implements *java.awt.event.ActionListener*

Das Fenster mit den Übersetzungsvorschlägen beim Kopieren

Konstruktoren

+ TranslationWindow ()
  Konstruktor

Methoden

+ setTables (*Hashtable, Hashtable, Hashtable*) → *void*
  Die Übersetzungstabellen setzen
+ getZustandTable () → *Hashtable*
  Die Zustandstabelle zurückgeben
+ getEreignisTable () → *Hashtable*
  Die Ereignistabelle zurückgeben
+ getVariableTable () → *Hashtable*
  Die Variablentabelle zurückgeben
+ setTopState (*GrafZustand*) → *void*
  Den topState der Translateion setzen
+ show () → *void*
  Das Fenster zeichnen
– dispose () → *void*
  Das Fenster löschen
+ actionPerformed (*ActionEvent*) → *void*
  Auf den OK-Knopf reagieren

D.14 Package score.SimSCM

D.14.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.LogSCM.LogPunkt*
    ↪ *score.SimSCM.SimBasic*
      ↪ *score.SimSCM.SimAND*
      ↪ *score.SimSCM.SimOR*
    ↪ *score.SimSCM.SimConn*
      ↪ *score.SimSCM.SimHist*
  ↪ *score.SimSCM.SimSCModell*

D.14.2 Exception NoTargetStateException

extends *java.lang.Exception*
Diese Exception wird geworfen, wenn beim Aktivieren eines Zustandes kein zu aktivierende Unterzustand festgestellt werden kann, oder wenn der Oberzustand weder OR- noch AND-Zustand ist.

Konstruktoren

\+ NoTargetStateException (*String*)
Erzeugt eine neue Exception mit dem angegebenen Text.

D.14.3 Klasse SimAND

extends *score.SimSCM.SimBasic*
Diese Klasse entspricht in der Simulation einem logischen AND-Zustand.

Konstruktoren

\+ SimAND (*ZustAND, SimBasic*)
Erschafft einen neuen AND-Simulatinspunkt aus dem übergebenen logischen AND-Zustand. Der Oberzustand wird dem übergebenen Identifier entsprechend gesetzt.

\+ SimAND (*int, int*)
Erschafft einen neuen AND-Simulationspunkt mit Hilfe von zwei Identifiern

Methoden

+$ prepareTimestep () → *void*
Bereitet einen Zeitschritt vor.

+$ closeTimestep () → *void*
Beendet einen Zeitschritt. Dabei werden in allen AND-Simulationspunkten, die aktiviert wurden, diejenigen Partitionen aktiviert, die noch nicht als zukünftig aktiv markiert sind.

\+ activate () → *void*
überschreibt die Methode der Klasse *SimBasic*.
Aktiviert diesen AND-Simulationspunkt. Dabei werden auch alle Partitionen aktiviert.

– completeActivation () → *void*
Vervollständigt die Aktivierung dieses AND-Simulationspunktes. Dabei werden alle Partitionen, die noch nicht als zukünftig aktiv markiert sind, jetzt aktiviert.

\+ activateByHistory () → *void*
überschreibt die Methode der Klasse *SimBasic*.
Wirft eine NoTargetStateException, da ein AND-Zustand nicht mittels eines History-Einstiegs aktiviert werden kann.

\+ activateByDeepHistory () → *void*

überschreibt die Methode der Klasse *SimBasic*.
Aktiviert diesen AND-Simulationspunkt (bzw. alle seine Partitionen) Deep-History-Einstieg.

+ activateUpwards (*SimBasic*) → *void*

überschreibt die Methode der Klasse *SimBasic*.
Aktiviert diesen AND-Simulationspunkt im aufsteigenden Durchlauf. Er wird dabei in die Liste der nachzuführenden AND-Zustände eingetragen. Diese Methode sollte nicht von außen aufgerufen werden.

+ deactivate () → *void*

überschreibt die Methode der Klasse *SimBasic*.
Deaktiviert diesen AND-Simulationspunkt und alle seine Partitionen.

+ deactivateBelow (*int*) → *void*

überschreibt die Methode der Klasse *SimBasic*.
Deaktiviert alle Simulationszustände unterhalb der Simulationsentsprechung des übergebenen Zustands.

+ deepClearHistory () → *void*

überschreibt die Methode der Klasse *SimBasic*.
Löscht die History aller Unterzustände.

+ addSubstate (*SimBasic*) → *void*

überschreibt die Methode der Klasse *SimBasic*.
Fügt einen SimBasic als Partition hinzu.

D.14.4 Klasse SimBasic

extends *score.LogSCM.LogPunkt*
implements *java.lang.Comparable*

Diese Klasse entspricht in der Simulation einem logischen atomaren Zustand.

Attribute

+$ active:*Vector* – statischer Vector mit allen SimBasics, die gerade aktiv sind

+$ willBeActive:*Vector* – statischer Vector mit allen SimBasics, die derart markiert sind, dass sie im nächsten Zeitschritt aktiv sind.

+$ calledActions:*Vector* – statischer Vector mit allen CalledActions, die durch Aktivieren und Deaktivieren von SimBasics ausgelöst wurden.

\# state:*ZustandAtom* – ZustandAtom, aus dem dieser SimBasic erzeugt wurde.

\# superState:*SimBasic* – der Oberzustand der Simulation für diesen SimBasic

Konstruktoren

+ SimBasic (*ZustandAtom, SimBasic*)

Erschafft einen neuen atomaren Simulationspunkt zu dem übergebenen atomaren Zustand und mit dem übergebenen Identifier des Oberzustands.

+ SimBasic (*int, int, boolean*)

Erschafft einen SimBasic für das auführbare Programm.

Methoden

+ isEndState () → *boolean*

Gibt zurück, ob dieser SimBasic ein Endzustand ist.

+ getLogic () → *ZustandAtom*

Gibt den logischen Zustand zurück, aus dem dieser SimBasic erschaffen wurde.

+ getIncoming () → *Vector*

überschreibt die Methode der Klasse *LogPunkt*.
Gibt einen Vector mit allen eingehenden LogPfeilen zurück.

\+ getOutgoing () → *Vector*

überschreibt die Methode der Klasse *LogPunkt*.
Gibt einen Vector mit allen ausgehenden LogPfeilen zurück.

\+ getSuper () → *LogPunkt*

überschreibt die Methode der Klasse *LogPunkt*.
Gibt den direkten Überzustand dieses SimBasic zurück.

+$ getSimBasic (*int*) → *SimBasic*

Gibt den Simulationspunkt mit dem übergebenen Identifier zurück.

+$ getAllSimBasic () → *Collection*

Gibt eine Collection mit allen Simulationspunkten zurück.

+$ prepareTimestep () → *void*

Bereitet einen Zeitschritt vor. Dabei werden alle momentan aktiven SimBasic so markiert, dass sie auch im nächsten Zeitschritt aktiviert sind. Außerdem wird der Vector der CalledActions geleert.

+$ closeTimestep () → *void*

Beendet einen Zeitschritt. Dabei werden die SimBasics, die markiert wurden, jetzt auch aktiviert.

\# activateBasic () → *void*

Aktiviert diesen Simulationspunkt. Er wird in die Liste der zukünftig aktiven Punkte eingetragen. Außerdem wird das Ereignis, das beim Aktivieren dieses Punktes ausgelöst wird, in die Ereignis-Menge eingetragen. Ober- oder Unterzustände werden nicht beachtet. Diese Methode sollte nur von innerhalb des Simulationspunktes selbst aufgerufen werden.

\# deactivateBasic () → *void*

Deaktiviert diesen Simulationspunkt. Er wird aus der Liste der zukünftig aktiven Punkte gestrichen. Außerdem wird das Ereignis, das beim Deaktivieren dieses Punktes ausgelöst wird, in die EreignisMenge eingetragen. Ober- oder Unterzustände werden nicht beachtet. Diese Methode sollte nur von innerhalb des Simulationspunktes selbst aufgerufen werden.

\+ isActive () → *boolean*

Gibt zurück, ob dieser Simulationspunkt im Moment aktiv ist.

\+ hasHistory () → *boolean*

Gibt zurück, ob dieser SimBasic eine History hat – solange diese Methode nicht von einem SimOR überschrieben wird, gibt sie immer `false` zurück.

\+ willBeActive () → *boolean*

Gibt zurück, ob dieser Simulationspunkt in die Liste der zukünftig aktiven Punkte eingetragen ist.

\+ activate () → *void*

Aktiviert diesen atomaren Simulationspunkt, wenn er noch nicht als zukünfig aktiv markiert ist. Der aufsteigende Durchlauf wird begonnen. Ansonsten wird eine NoTargetStateException geworfen.

\+ activateByHistory () → *void*

Wirft eine NoTargetStateException, da atomare Zustände nicht über einen History-Einstieg betreten werden können.

\+ activateByDeepHistory () → *void*

Aktiviert diesen Simulationspunkt, wenn er noch nicht als zukünftig aktiv markiert ist. Der aufsteigende Durchlauf wird begonnen.

\+ activateUpwards (*SimBasic*) → *void*

Wirft eine NoTargetStateException, da ein atomarer Zustand nicht von unten aktiviert werden kann.

\+ deactivate () → *void*

Deaktiviert diesen atomaren Simulationspunkt.

\+ deactivateBelow (*int*) → *void*

Deaktiviert diesen Simulationspunkt und alle ihm übergeordneten, die sich noch unterhalb des angegebenen Zustands befinden.

+ deactivateUpwards (*int*) → *void*

  Deaktiviert diesen SimBasic und alle seine Oberzustände unterhalb der Simulationsentsprechung des übergebenen Zustands.

+ toggleActivation () → *boolean*

  Schaltet den Zustand von aktiviert auf nicht aktiviert um oder umgekehrt. Gibt `false` zurück, wenn dabei eine NoTargetStateException geworfen wurde.

+ deepClearHistory () → *void*

  Löscht die History aller Unterzustände. Muss entsprechend implementiert werden.

+ setActionsActivate (*AktKomplex*) → *void*

  Setzt die Aktionen beim Aktivieren dieses SimBasic. Diese Methode wird nur im exportierten Quelltext benötigt.

+ setActionsDeactivate (*AktKomplex*) → *void*

  Setzt die Aktionen beim Deaktivieren dieses SimBasic. Diese Methode wird nur im exportierten Quelltext benötigt.

+ addSubstate (*SimBasic*) → *void*

  Fügt einen Unterzustand hinzu - solange diese Methode nicht überschrieben wird, gibt sie einen Fehler aus, da zu einem SimBasic keine Unterzustände existieren können.

+ compareTo (*Object*) → *int*

  Implementation des Comparable-Interfaces; akzeptiert nur andere SimBasics

D.14.5 Klasse SimConn

extends *score.LogSCM.LogPunkt*
Diese Klasse stellt einen Konnektor für das ausführbare Programm dar. In der Simulation wird sie nicht benötigt, da dort der LogPunkt selbst benutzt wird.

Konstruktoren

+ SimConn (*int, int*)

  Erschafft einen Universalkonnektor

Methoden

+ getSuper () → *LogPunkt*

  überschreibt die Methode der Klasse *LogPunkt*.
  Gibt den LogPunkt zurück, in dem dieser Konnektor liegt.

D.14.6 Klasse SimHist

extends *score.SimSCM.SimConn*
Diese Klasse erweitert den Konnektor für das exportierte Programm im Sinne eines History-Konnektors, d. h. um die Angabe, ob es sich um eine Deep- oder FlatHistory handelt.

Konstruktoren

+ SimHist (*int, int, boolean*)

  Erschafft einen History-Konnektor mit den angegebenen Eigenschaften.

D.14.7 Klasse SimOR

extends *score.SimSCM.SimBasic*
Diese Klasse entspricht in der Simulation einem OR-Zustand oder einer Partition.

Konstruktoren

+ SimOR (*ZustOR, SimBasic*)

  Erschafft einen neuen OR-Simulationspunkt aus dem übergebenen logischen OR-Zustand. Er hat den übergebenen Zustand als Oberzustand. Der Default-Einstieg wird entsprechend gesetzt, falls der logische Zustand einen besitzt; die History ist noch leer.

+ SimOR (*int, int*)

  Erschafft einen OR-Simulationspunkt aus zwei Identifiern.

Methoden

+ addNewSub (*ZustandAtom*) → *SimBasic*

  Fügt einen neuen Unterzustand hinzu, der aus dem übergebenen atomaren Zustand generiert wird.

+ activate () → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Aktiviert diesen OR-Simulationspunkt. Dabei wird der Default-Einstieg benutzt.

+ activateByHistory () → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Aktiviert diesen OR-Simulationspunkt durch seinen History-Einstieg. Es wird davon ausgegangen, dass tatsächlich eine History vorhanden ist, ansonsten wird eine NoTargetStateException geworfen.

+ activateByDeepHistory () → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Aktiviert diesen OR-Simulationspunkt durch einen Deep-History-Einstieg. Ist keine History gesetzt, wird der Default-Einstieg benutzt. Der ausgewählte Unterzustand wird ebenfalls mittles Deep-History-Einstieg aktiviert.

+ activateUpwards (*SimBasic*) → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Aktiviert diesen OR-Simulationspunkt im aufsteigenden Durchlauf. Diese Methode sollte nicht von außen aufgerufen werden. Sie übernimmt das eigentliche Aktivieren des Simulationspunktes und fährt dann bei seinem Oberzustand fort, solange dieser noch nicht markiert ist.

+ deactivate () → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Deaktiviert diesen Zustand und den als aktiv gemerkten Unterzustand.

+ deactivateBelow (*int*) → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Deaktiviert diesen Zustand und alle Oberzustände unterhalb des übergebenen Zustands.

+ hasHistory () → *boolean*

  überschreibt die Methode der Klasse *SimBasic*.
  Gibt zurück, ob dieser Zustand momentan eine nicht-leere History besitzt.

+ clearHistory () → *void*

  Löscht die History dieses Zustands.

+ deepClearHistory () → *void*

  überschreibt die Methode der Klasse *SimBasic*.
  Löscht die History dieses Zustands und aller Unterzustände.

+ awaitDefault (*int*) → *void*

Gibt diesem OR-Zustand den Identifier des Default-Einstiegs. Dieser ist jedoch noch nicht als Unterzustand eingefügt; wenn dies geschieht, wird er dann automatisch als Default-Einstieg markiert.

+ addSubstate (*SimBasic*) → *void*

überschreibt die Methode der Klasse *SimBasic*.
Fügt einen Unterzustand hinzu.

D.14.8 Klasse SimSCModell

extends *java.lang.Object*
Dies ist die zentrale Klasse des Simulationsmodells. Sie implementiert das Interface ConditionAskable, mit dem sich Bedingungen nach ihrem Wahrheitswert in der aktuellen Simulation erkundigen können.

Konstruktoren

+ SimSCModell (*LogSCModell*)

Erschafft ein neues Simulationsmodell, das dem übergebenen logischen Modell entspricht. Jeder Zustand existiert in einfacher Ausführung.

+ SimSCModell ()

Erschafft ein neues Simulationsmodell, das davon ausgeht, dass sein Oberzustand der (bereits existierende) SimBasic mit dem Identifier `0` ist.

Methoden

+ getTopLevelState () → *SimOR*

Gibt den Oberzustand der Simulation zurück.

+ prepareTimestep () → *void*

Bereitet einen Zeitschritt im Simulationsmodell vor. Dabei wird insbesondere die Liste der zukünftig zu aktivierenden Simulationspunkte als Liste der momentan aktivierten Punkte übernommen.

+ closeTimestep () → *boolean*

Schließt einen Zeitschritt im Simulationsmodell ab. Gibt `true` zurück, wenn das ohne Fehler möglich war.

+ getSimBasic (*int*) → *SimBasic*

Implementation des ConditionAskable-Interfaces. Gibt den SimBasic mit dem angegeben Identifier zurück.

+ activate (*int*) → *void*

Aktiviert den SimBasic mit dem angegeben Identifier. Wirft eine NoTargetStateException, wenn dabei Fehler auftreten.

+ deactivate (*int*) → *void*

Deaktiviert den SimBasic mit dem angegebenen identifier.

+ getAllActive () → *Vector*

Gibt einen Vector aller Simulationspunkte zurück, die für diesen Zeitschritt aktiv sind.

+ getAllFutureActive () → *Vector*

Gibt einen Vector aller Simulationspunkte zurück, die für den nächsten Zeitschritt als aktiv markiert sind.

+ getCalledActions () → *Vector*

Gibt den Vector aller Aktionen zurück, die im letzten Zeitschritt aufgerufen wurden.

D.15 Package score.Simulator

D.15.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.AbstractModell.MarkableTyp*
    ↪ *score.Simulator.Variable*
      ↪ *score.Simulator.SimVariable*
  ↪ *score.Simulator.SCRun*
    ↪ *score.Simulator.Simulation*
  ↪ *score.Simulator.SimWindow*

D.15.2 Interface VarServer

Dieses Interface regelt den Zugriff um eine Variable zu erhalten

Methoden

+ getVariable (*String*) → *Variable*
  Liefert eine Variable mit Namen varName

D.15.3 Klasse SCRun

(abstrakte Klasse)
extends *java.lang.Object*
implements *java.awt.event.ActionListener,*
java.lang.Runnable,
score.Simulator.VarServer,
score.Bedingungen.ConditionAskable,
score.Aktionen.EventHolder

Diese Klasse ist die abstrakte Oberklasse aller Klassen, die ein Statechart ausführen wollen i. e. der Simulator und der Export

Attribute

delay:*int* – Wartezeit um auf Benutzeraktionen zu warten
simModell:*SimSCModell* – Referenz auf das Simulationsmodell
time:*int* – Die Simulationszeit
phase:*int* – Die Simulationsphase
events:*Vector* – Die gesammelten Events
actions:*Vector* – Die gesammelten Aktionen
variables:*Hashtable* – Die Variablen um als VarServer zu arbeiten
thread:*Thread* – Der Thread um Nebenläufig zu laufen
phaseFinished:*boolean* – Flag ob eine Phase abgeschlossen ist
running:*boolean* – Flag ob die Simulation gerade läuft
logger:*Logger* – Eine Referenz auf einen Logger um Meldungen auszugeben

Konstruktoren

+ SCRun ()
  Konstruktor (macht nix)

Methoden

init (*LogSCModell*) → *void*

Das Simulationsmodell erstellen und die Ressourcen anfordern

\# generateSCModell (*LogSCModell*) → *SimSCModell*
Das Simulationsmodell aus einem logischen Modell erzeugen

\+ isActive (*LogPunkt*) → *boolean*
Prüfen ob ein Zustand lp aktiv ist

\# getSimBasic (*LogPunkt*) → *SimBasic*
Liefert den SimBasic zu einem logPunkt

\+ getSimBasic (*int*) → *SimBasic*
Implementation des Condition Askable Interfaces

\# setPhase (*int*) → *void*
Die Phase setzen

\+ stepPhase () → *void*
Eine Phase weiter setzen

\# start () → *void*
Den Ablauf starten (Thread erzeugen)

\# stop () → *void*
Den Ablauf stoppen

\+ run () → *void*
Der Ablauf um nebenläufigen Thread

\+ startSimulation () → *void*
Die Simulation starten

\+ stopSimulation () → *void*
Die Simulation stoppen

\# runPhase () → *void*
Ein Phase abarbeiten

\# resetTime () → *void*
Die Zeit auf 0 setzen

\# incTime () → *void*
Die Zeit erhöhen

\# doTarget (*LogPunkt, Vector, Vector*) → *boolean*
Ein Ziel eines Übergangs behandeln

\# doTransitions () → *void*
Die Übergänge durchführen

\# checkTransition (*LogPfeil, Vector, Vector*) → *boolean*
Übergänge prüfen

\# checkForEndStates () → *void*
Prüfen ob ein EndState erreicht wurde

\+ getEvents () → *Vector*
Die Events liefern

\+ storeEvent (*Object*) → *void*
Ein Event speichern

\# createDialogs () → *void*
Die Dialoge erschaffen

\# fetchDlgActions () → *void*
Die Aktionen der Dialoge holen

D.15.4 Klasse Simulation

extends *score.Simulator.SCRun*
implements *score.Ereignisse.EventAskable,*
score.util.Observer,
score.Bedingungen.ConditionAskable,
javax.swing.event.ListSelectionListener,
java.awt.event.WindowListener

Diese Klasse implementiert SCRun für die Simulation

Konstruktoren

\+ Simulation (*GrafSCModell, LogDModell, Observable*)
Konstruktor

Methoden

\# setPhase (*int*) → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Phase setzen

\# createDialogs () → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Dialoge erzeugen

\+ occurred (*Ereignis*) → *boolean*
Prüfen ob Ereignis e geschehen ist

\+ refreshLists () → *void*
Setze die Listen neu

\# runPhase () → *void*
überschreibt die Methode der Klasse *SCRun*.
Eine Phase bearbeiten

\# resetTime () → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Zeit auf 0 setzen

\# incTime () → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Zeit erhöhen

\+ startSimulation () → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Simulation starten

\+ stopSimulation () → *void*
überschreibt die Methode der Klasse *SCRun*.
Die Simulation stoppen

\+ actionPerformed (*ActionEvent*) → *void*
Auf die ButtonEvents reagieren

\# fetchDlgActions () → *void*
überschreibt die Methode der Klasse *SCRun*.
Eine Dialogaktion holen

\+ update (*Observable, Object*) → *void*
Aktion von Dialogkomponente bekommen

\+ getSimBasic (*int*) → *SimBasic*
überschreibt die Methode der Klasse *SCRun*.
ConditionAskable Interface

\+ getVariable (*String*) → *Variable*
VarServer Interface

D.15.5 Klasse SimVariable

extends *score.Simulator.Variable*
Diese Klasse implementiert eine Variable für die Simulation

Konstruktoren

+ SimVariable (*String*)
  Konstruktor
+ SimVariable (*String, String*)
  Konstruktor
+ SimVariable (*String, int*)
  Konstruktor
+ SimVariable (*String, double*)
  Konstruktor

Methoden

checkDialogVar () → *void*
  überschreibt die Methode der Klasse *Variable*.
  Prüft ob dies eine Dialogvariable ist und holt von dort einen Wert

D.15.6 Klasse SimWindow

extends *java.lang.Object*
implements *score.control.Logger*

Diese Klasse stellt die grafischen Komponenten des Simulationsfensters

Attribute

+ logList:*JList* – Die verschiedenen Listen in der Anzeige
+ activeStateList:*JList* – Die verschiedenen Listen in der Anzeige
+ actionList:*JList* – Die verschiedenen Listen in der Anzeige
+ variableList:*JList* – Die verschiedenen Listen in der Anzeige
+ eventList:*JList* – Die verschiedenen Listen in der Anzeige
+ delay:*NumField* – Das Zahlenfeld für die Zeit

Konstruktoren

+ SimWindow (*ActionListener, WindowListener*)
  Konstruktor

Methoden

+ setTime (*int*) → *void*
  Die Zeit setzen
+ setPhase (*String*) → *void*
  Die Phase setzen
+ startRunning () → *void*
  Auszuführen beim Start der Simulation
+ stopRunning () → *void*
  Auszuführen beim Ende der Simulation
+ addLine (*String*) → *void*
  Eine Meldung ausgeben
+ dispose () → *void*
  Das Fenster löschen

D.15.7 Klasse Variable

(abstrakte Klasse)
extends *score.AbstractModell.MarkableTyp*
implements *java.lang.Comparable*

Dies ist die abstrakte Oberklasse für die Variablen

Attribute

+$ compServer:*DialogCompServer* – Referenz auf den DialogkomponentenServer

\# name:*String* – Der Name der Variablen

+$ DLG_PREFIX:*String* CONST – Das Prefix für Dialogvariablen

+$ LESS:*int* CONST – Konstanten für die Vergleichsoperatoren

+$ GREATER:*int* CONST – Konstanten für die Vergleichsoperatoren

+$ EQUAL:*int* CONST – Konstanten für die Vergleichsoperatoren

+$ UNEQUAL:*int* CONST – Konstanten für die Vergleichsoperatoren

+$ LESS_OR_EQUAL:*int* CONST – Konstanten für die Vergleichsoperatoren

+$ GREATER_OR_EQUAL:*int* CONST – Konstanten für die Vergleichsoperatoren

Konstruktoren

\+ Variable (*String*)
Konstruktor

\+ Variable (*String, String*)
Konstruktor

\+ Variable (*String, int*)
Konstruktor

\+ Variable (*String, double*)
Konstruktor

Methoden

\# checkDialogVar () → *void*
Prüfen, ob dies eine Dialogvariable ist

\+ inc (*String*) → *void*
Variable erhöhen

\+ inc (*int*) → *void*
Variable erhöhen

\+ inc (*double*) → *void*
Variable erhöhen

\+ set (*String*) → *void*
Variable setzen

\+ set (*int*) → *void*
Variable setzen

\+ set (*double*) → *void*
Variable setzen

\+ inc (*Variable*) → *void*
Variable erhöhen

\+ set (*Variable*) → *void*

Variable setzen

+ getName () → *String*
  Den Namen der Variablen auslesen
+ getString () → *String*
  String-Repräsentation der Variablen liefern
+ getDouble () → *double*
  Double-Repräsentation der Variablen liefern
+ getInt () → *int*
  Integer-Repräsentation der Variablen liefern
+ compare (*int*, *int*) → *boolean*
  Variable vergleichen
+ compare (*int*, *String*) → *boolean*
  Variable vergleichen
+ compare (*int*, *double*) → *boolean*
  Variable vergleichen
+ compare (*int*, *Variable*) → *boolean*
  Variable vergleichen
+ compareTo (*Object*) → *int*
  Variable vergleichen
+ toString () → *String*
  überschreibt die Methode der Klasse *MarkableTyp*.
  Variable als String darstellen

D.16 Package score.util

D.16.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *java.awt.Dimension*
    ↪ *score.util.Dimension*
  ↪ *java.awt.geom.Dimension2D*
    ↪ *score.util.Dimension.Double*
  ↪ *java.util.Stack*
    ↪ *score.util.UndoStack*
  ↪ *javax.swing.JButton*
    ↪ *score.util.AnimatedImage*
    ↪ *score.util.FlatButton*
  ↪ *javax.swing.JLabel*
    ↪ *score.util.BlackLabel*
  ↪ *javax.swing.JPanel*
    ↪ *score.util.NumField*
  ↪ *javax.swing.JRadioButton*
    ↪ *score.util.FlatRadioButton*
  ↪ *javax.swing.JTextField*
    ↪ *score.util.TextEditField*
  ↪ *score.util.Compare*
  ↪ *score.util.EasyFileWriter*
    ↪ *score.util.EasyStringWriter*
  ↪ *score.util.Intersect*
  ↪ *score.util.LaTeXString*
  ↪ *score.util.Observable*
  ↪ *score.util.Pair*

↪ *score.util.Scale*
↪ *score.util.StringCheck*
↪ *score.util.Triplet*
↪ *score.util.VecArithmetik*
↪ *score.util.ZoomGraphics*
↪ *score.util.MetaPostGraphics*

D.16.2 Interface ErrorPrinter

Dies ist ein Interface um Fehlermeldungen auszugeben

Methoden

+ printError (*String*) → *void*
  Gibt die Fehlermeldung aus

D.16.3 Interface Observer

Dieses Interface entspricht dem Java-Interface Observer. Es bezieht sich allerdings auf eine andere Observable-Klasse, nämlich `score.util.Observable`. Zudem hat es die Fähigkeit, ein bestimmtes beobachtetes Objekt durch ein anderes zu ersetzen. Bei der Implementation ist darauf zu achten, dass die Ersetzung auch durch den Wert `Null` geschehen kann; dies heißt, dass das beobachtete Objekt gelöscht wurde.

Methoden

+ update (*Observable, Object*) → *void*
  Empfängt ein update vom beobachteten Objekt, in dem ein notifyObservers mit dem hier übergebenen Argument aufgerufen wurde.
+ replaceObservable (*Observable, Observable*) → *boolean*
  Ersetzt das übergebene beobactete Objekt durch ein anderes. Eine Ersetzung durch den Wert `Null` bedeutet, dass das Objekt gelöscht wurde. Es ist nicht notwendig, die entsprechenden Aufrufe der Methode addObserver bzw. deleteObserver in den Observables aufzurufen, sondern nur eventuelle zusätzliche Aktionen, die dieser Observer auszuführen hat.

D.16.4 Interface UndoManager

Dieses Interface stellt die Funktionen für das Rückgängig machen von Editiervorgängen zur Verfügung

Methoden

+ undo () → *void*
  Einen Schritt Rückgängig machen
+ prepare () → *void*
  Aktuellen Zustand merken
+ commit () → *void*
  Aktuellen Zustand sichern zum Rückgängig machen
+ isPrepared () → *boolean*
  Ist ein aktueller Zustand gemerkt?

D.16.5 Klasse AnimatedImage

extends *javax.swing.JButton*
implements *java.lang.Runnable*

Diese Klasse stellt ein animiertes Bild dar

Konstruktoren

+ AnimatedImage ()
  Konstruktor
+ AnimatedImage (*String*)
  Konstruktor
+ AnimatedImage (*ImageIcon*)
  Konstruktor

Methoden

+ setDelay (*int*) → *void*
  Die Verzögerung zwischen zwei Bildern setzen
+ reset () → *void*
  Die Animation resetten
+ start () → *void*
  Die Animation starten
+ stop () → *void*
  Die Animation stoppen
+ run () → *void*
  Die Animation läuft
+ addImage (*String*) → *void*
  Ein Bild hinzufügen
+ addImage (*ImageIcon*) → *void*
  Ein Bild hinzufügen

D.16.6 Klasse BlackLabel

extends *javax.swing.JLabel*
Stellt einen speziell Konfigurierten Label dar

Konstruktoren

+ BlackLabel (*String*)
  Konstruktor
+ BlackLabel (*String, int*)
  Konstruktor

D.16.7 Klasse Compare

extends *java.lang.Object*
Diese Klasse enthält eine Vergleichsfunktion

Konstruktoren

+ Compare ()
  Default-Konstruktor.

Methoden

+$ isSimilar (*Point, Point, int*) → *boolean*
  Prüft ob zwei Punkte a und b mit einer Toleranz tolerance gleich sind

D.16.8 Klasse Dimension

extends *java.awt.Dimension*
innere Klassen: *Dimension.Double*
Diese Klasse erweiter java.awt.Dimension um Double-Werte wie Point2D.Double

Konstruktoren

+ Dimension ()
  Konstruktor
+ Dimension (*Dimension*)
  Konstruktor
+ Dimension (*int, int*)
  Konstruktor

D.16.9 Klasse Dimension.Double

extends *java.awt.geom.Dimension2D*
Innere Klasse für einen Double-Dimensions Wert

Attribute

+ width:*double* – Breite
+ height:*double* – Höhe

Konstruktoren

+ Dimension.Double ()
  Konstruktor (tut nix)
+ Dimension.Double (*double, double*)
  Konstruktor
+ Dimension.Double (*Dimension.Double*)
  Copy-Konstruktor

Methoden

+ getHeight () → *double*
  überschreibt die Methode der Klasse *Dimension2D*.
  Liefert die Höhe
+ getWidth () → *double*
  überschreibt die Methode der Klasse *Dimension2D*.
  Liefert die Breite
+ equals (*Dimension.Double*) → *boolean*
  Prüft auf Gleichheit
+ setSize (*Dimension.Double*) → *void*
  Setzt die Größe neu
+ setSize (*double, double*) → *void*
  überschreibt die Methode der Klasse *Dimension2D*.
  Setzt die Größe neu
+ getLength () → *double*
  Liefert die Länge (als Vektor interpretiert)
+$ normalize (*Dimension.Double*) → *Dimension.Double*
  Normalisiert (als Vektor)
+ normalize () → *Dimension.Double*
  Normalisiert (als Vektor)

D.16.10 Klasse EasyFileWriter

extends *java.lang.Object*
Diese Klasse erweitert im Prinzip den FileWriter durch mehrere Schreibmethoden, die verschiedene Typen akzeptieren. Tatsächlich wird der FileWriter intern gehalten, damit der EasyFileWriter durch den EasyStringWriter erweitert werden kann.

Attribute

#$ newline:*String* `CONST` – die Systemabhängige Konstante für einen Zeilenumbruch

Konstruktoren

\+ EasyFileWriter (*String*)
Erschafft einen neuen EasyFileWriter, der in die Datei mit dem angegebenen Namen schreiben wird.

\+ EasyFileWriter (*File*)
Erschafft einen neuen EasyFileWriter, der in das übergebene File schreiben wird.

\# EasyFileWriter ()
Erschafft einen neuen EasyFileWriter ohne FileWriter. Dies ist nur sinnvoll, wenn die Schreibmethoden entsprechend überladen werden.

Methoden

\+ isFileWriter () → *boolean*
Gibt zurück, ob es sich tatsächlich um einen EasyFileWriter handelt.

\# doWrite (*String*) → *void*
Führt das eigentliche Schreiben aus. Der übergebene String wird durch ein angehängtes Leerzeichen ergänzt.

\+ writeKey (*String*) → *void*
Schreibt den übergebenen String unmodifizert in die Datei.

\+ writeKeyLn (*String*) → *void*
Schreibt den übergebenen String unmodifiziert in die Datei und hängt einen Zeilenumbruch an.

\+ write (*String*) → *void*
Schreibt den übergebenen String mit Anführungszeichen in die Datei.

\+ write (*boolean*) → *void*
Schreibt den übergebenen booleschen Wert in die Datei.

\+ write (*int*) → *void*
Schreibt den übergebenen int-Wert in die Datei.

\+ write (*double*) → *void*
Schreibt den übergebenen double-Wert in die Datei.

\+ write (*Point2D.Double*) → *void*
Schreibt den übergebenen Point2D.Double in die Datei. Er wird dabei mit runden Klammern und einem Komma formatiert.

\+ write (*Dimension.Double*) → *void*
Schreibt die übergebene Dimension.Double in die Datei. Sie wird dabei mit eckigen Klammern und einem Komma formatiert.

\+ writeln () → *void*
Schreibt einen Zeilenumbruch in die Datei.

\+ insertFile (*String*) → *void*
Fügt den Inhalt einer Datei ein.

\+ close () → *void*
Schließt den FileWriter.

\+ flush () → *void*
Gibt ein flush() an den FileWriter weiter.

D.16.11 Klasse EasyStringWriter

extends *score.util.EasyFileWriter*
Diese Klasse erweitert den EasyFileWriter derart, dass er nicht in ein File, sondern in einen String schreibt.

Konstruktoren

+ EasyStringWriter ()
  Erschafft einen neuen EasyStringWriter mit einem leeren Buffer.

Methoden

+ isFileWriter () → *boolean*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Gibt zurück, dass es sich nicht um einen EasyFileWriter handelt.
doWrite (*String*) → *void*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Schreibt den übergebenen String in den Buffer und hängt ein Leerzeichen an.
+ writeln () → *void*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Schreibt einen Zeilenumbruch in den Buffer.
+ insertFile (*String*) → *void*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Fügt ein File mit dem angegebenen Namen ein. Diese Methode ist noch nicht implementiert.
+ getString () → *String*
  Gibt den erzeugten String zurück.
+ close () → *void*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Setzt die Methode des EasyFileWriters außer Kraft.
+ flush () → *void*
  überschreibt die Methode der Klasse *EasyFileWriter*.
  Setzt die Methode des EasyFileWriters außer Kraft.

D.16.12 Klasse FlatButton

extends *javax.swing.JButton*
implements *java.awt.event.MouseListener*

Diese Klasse erweitert JButton derart, dass die Buttons flach aussehen und beim Überqueren oder Selektieren mit der Maus ihr Image ändern

Konstruktoren

+ FlatButton (*ImageIcon, ImageIcon, ImageIcon*)
  Konstruktor

D.16.13 Klasse FlatRadioButton

extends *javax.swing.JRadioButton*
implements *java.awt.event.MouseListener*

Diese Klasse erweitert JRadioButton analog zu FlatButton

Konstruktoren

+ FlatRadioButton (*ImageIcon, ImageIcon, ImageIcon*)

Konstruktor

Methoden

+ doUnclick () → *void*
  Führt ein deselektieren durch

+ doClick () → *void*
  überschreibt die Methode der Klasse *AbstractButton*.
  Führt das Selektieren durch

D.16.14 Klasse Intersect

extends *java.lang.Object*
Diese Klasse testet den Schnitt von Strecken bzw. Rechtecken in zwei Formaten.

Methoden

+$ lines (*double, double, double, double, double, double*) → *boolean*
  Gibt `true` zurück, wenn sich die horizontale und die vertikale Strecke schneiden (oder berühren).

+$ rectangles (*Point2D.Double, Point2D.Double, Point2D.Double, Point2D.Double*) → *boolean*
  Gibt `true` zurück, wenn sich die beiden Rechtecke (a1/a2) und (b1/b2) schneiden (oder berühren).

+$ rectangles (*double, double, double, double, double, double, double, double*) → *boolean*
  Gibt zurück, ob sich die beiden Rechtecke (gegeben in jeweils vier double-Koordinaten, schneiden (oder berühren).

D.16.15 Klasse LaTeXString

extends *java.lang.Object*
Diese Klasse dient dazu, einen String mit deutschen Umlauten in TeX-Format umzuschreiben.

Methoden

+$ prepareForMetaPost (*String*) → *String*
  Bereitet den übergebenen String zum Einbinden in MetaPost vor.

+$ convert (*String*) → *String*
  Konvertiert den übergebenen String nach TeX.

D.16.16 Klasse MetaPostGraphics

extends *score.util.ZoomGraphics*
Diese Klasse überschreibt die Klasse *ZoomGraphics* derart, dass die Ausgabe nicht auf einem Grafikbildschirm erscheint, sondern in einem File in MetaPost-Format gespeichert wird. Dies geschieht in den `protected` Methoden mit dem Zusatz `Here`, die die entsprechenden Methoden von *ZoomGraphics* überschreiben, und bereits skalierte Punkte erwarten.

Konstruktoren

+ MetaPostGraphics (*double*)
  Erschafft eine neue MetaPostGraphics mit dem angegebenen Zoom.

Methoden

+ open (*String*) → *void*
  Öffnet das File mit dem übergenenen Namen zum schreiben.

+ close () → *void*
  Schließt das File, das in dieser MetaPostGraphics gehalten ist.

drawStringHere (*String, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet einen String an der übergebenen Stelle.

\# drawLineHere (*int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet eine Linie zwischen den übergebenen Punkten.

\# drawRectHere (*int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein Rechteck mit der übergebenen Position und Größe.

\# fillRectHere (*int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein ausgefülltes Rechteck mit der übergebenen Position und Größe.

\# drawRoundRectHere (*int, int, int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein Rechteck mit abgerundeten Ecken mit der übergebenen Position und Größe.

\# fillRoundRectHere (*int, int, int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein ausgefülltes Rechteck mit abgerundeten Ecke mit der übergebenen Position und Größe.

\# drawOvalHere (*int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein Oval mit der übergebenen Position und Größe.

\# fillOvalHere (*int, int, int, int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein ausgefülltes Oval mit der übergebenen Position und Größe.

\# fillPolygon (*int[], int[], int*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Zeichnet ein ausgefülltes Polygon mit den übergebenen Eckpunkten.

\+ drawActiveRect (*Point2D.Double, Dimension.Double*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Diese Methode lenkt das Zeichnen eines aktiven Rechtecks durch Überschreiben der entsprechenden Methode in *ZoomGraphics* auf das Zeichnen eines abgerundeten Rechtecks um.

\+ drawMarkedRect (*Point2D.Double, Dimension.Double*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Diese Methode lenkt das Zeichnen eines markierten Rechtecks durch Überschreiben der entsprechenden Methode in *ZoomGraphics* auf das Zeichnen eines abgerundenten Rechtecks um.

\+ drawHook (*Point2D.Double*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Deaktivert das Zeichnen von Hooks durch das Überschreiben der entsprechenden Methode in *ZoomGraphics*.

\+ setColor (*Color*) → *void*

überschreibt die Methode der Klasse *ZoomGraphics*.
Deaktiviert das Setzen der Farbe durch Überschreiben der entsprechenden Methode in *ZoomGraphics*.

\+ getColor () → *Color*

überschreibt die Methode der Klasse *ZoomGraphics*.
Gibt `Null` zurück, da diese Klasse keine Farben für MetaPost benutzt.

D.16.17 Klasse NumField

extends *javax.swing.JPanel*
implements *java.awt.event.ActionListener*

Diese Klasse zeigt ein Nummernfeld, mit Buttons zum de- und inkrementieren

Konstruktoren

+ NumField ()
  Konstruktor
+ NumField (*String, int*)
  Konstruktor
+ NumField (*String, int, int, int, int, int*)
  Konstruktor
+ NumField (*String, int, int, int, int*)
  Konstruktor
+ NumField (*String, int, int*)
  Konstruktor

Methoden

+ setNumber (*int*) → *void*
  Die Zahl setzen
+ setNumberSilent (*int*) → *void*
  Die Zahl Setzen ohne eine Event auszulösen
+ getNumber () → *int*
  Die Nummer auslösen
+ setStepSize (*int*) → *void*
  In- und Dekrement Schrittweite setzen
+ actionPerformed (*ActionEvent*) → *void*
  Die Nummer nach ButtonKlick setzen
+ addActionListener (*ActionListener*) → *void*
  Einen ActionListener eintragen
+ setActionCommand (*String*) → *void*
  Das ActionCommand setzen
+ removeActionListener (*ActionListener*) → *void*
  Einen ActionListener austragen

D.16.18 Klasse Observable

(abstrakte Klasse)
extends *java.lang.Object*
Diese abstrakte Klasse ersetzt die entsprechende Klasse `java.util.Observable` und erweitert sie um zusätzliche Zugriffsmethoden auf die Observer. Insbesondere ist dies die Methode replaceBy(), die dem Observer mitteilt, dass dieses beobachtete Objekt durch ein anderes zu ersetzen ist. Außerdem gibt es die Methode delete() die den Observern analog mitteilt, dass dieses Objekt zu entfernen ist.

Methoden

+ addObserver (*Observer*) → *void*
  Fügt einen Observer zu diesem Observable hinzu.
+ hasChanged () → *boolean*

Gibt zurück, ob sich dieses Observable geändert hat.

+ setChanged () → *void*
  Setzt die Angabe, dass sich dieses Observable geändert hat.
+ setObserversChanged () → *void*
  Setzt die Angabe, dass sich dieses Observable geändert hat. Für den Fall, dass ein Observer wieder Observable ist, wird dieser Aufruf in der Hierarchie nach oben durchgegeben.
clearChanged () → *void*
  Setzt die Angabe, dass sich dieses Observable nicht geändert hat.
+ countObservers () → *int*
  Gibt die Anzahl der registierten Observer zurück.
+ deleteObserver (*Observer*) → *void*
  Löscht einen Observer aus der Liste der registrierten Observer dieses Observables.
+ deleteObservers () → *void*
  Löscht alle Observer aus diesem Observable.
+ notifyObservers () → *void*
  Ruft in allen registrierten Observern die Methode update mit dem Argument `Null` auf.
+ notifyObservers (*Object*) → *void*
  Ruft in allen registrierten Observern die Methode update mit dem hier übergebenen Argument auf.
+ replaceBy (*Observable*) → *void*
  Ersetzt dieses Observable in allen Observern durch das übegebene Objekt. Wird dieses Obserable selbst übergeben, geschieht nichts; wird `Null` übergeben, bekommen die Observer den Aufruf, dass dieses Observable gelöscht wurde.
+ delete () → *void*
  Diese Methode teilt den Observern mit, dass dieses Objekt nicht länger existieren wird, und daher nicht mehr zu beobachten ist. Dies geschieht mittels des Aufrufs `replaceObservable(this, null)` im Observer. Die Observer werden jedoch nicht aus diesem Observable ausgetragen, und können daher auch später mit einem notifyObservers aufgerufen werden. Es wird hierbei NICHT auf die Methode `replaceBy` dieses Observables zugegriffen!

D.16.19 Klasse Pair

extends *java.lang.Object*
Diese Klasse hält ein geordnetes Paar von Instanzen beliebiger Klassen. Zugriffe erfolgen direkt auf die Instanzvariablen.

Attribute

+ first:*Object* – das erste Objekt
+ second:*Object* – das zweite Objekt

Konstruktoren

+ Pair (*Object*, *Object*)
  Erschafft ein geordnetes Paar mit den beiden übergebenen Objekten.

D.16.20 Klasse Scale

extends *java.lang.Object*
Diese statische Klasse ermöglicht das Umrechnen zwischen dargestellten und TapetenKoordinaten

Konstruktoren

+ Scale ()

Default-Konstruktor.

Methoden

+$ scale (*double, double*) → *int*
Skalieren

+$ unscale (*int, double*) → *double*
Rückwärts Skalieren

+$ unscale (*Point, double*) → *Point2D.Double*
Rückwärts Skalieren

+$ scale (*Point2D.Double, double*) → *Point*
Skalieren

+$ scale (*Dimension.Double, double*) → *Dimension*
Skalieren

+$ unscale (*Dimension, double*) → *Dimension.Double*
Rückwärts Skalieren

D.16.21 Klasse StringCheck

extends *java.lang.Object*
Diese Klasse stellt Routinen zur Verfügung um zu prüfen ob ein String eine Zahl ist

Konstruktoren

+ StringCheck ()
Default-Konstruktor.

Methoden

+$ isInt (*String*) → *boolean*
Ist der String s eine Integer-Zahl?

+$ isDouble (*String*) → *boolean*
Ist der String s eine Double-Zahl?

D.16.22 Klasse TextEditField

extends *javax.swing.JTextField*
Diese Klasse erweitert die Java-Klasse *java.swing.JTextField* derart, dass das Textfeld erkennt, ob vom Benutzer Änderungen vorgenommen wurden.

Konstruktoren

+ TextEditField ()
Erschafft ein neues Textfeld.

+ TextEditField (*Document, String, int*)
Erschafft ein neues Textfeld zum übergebenen Document und Text.

+ TextEditField (*int*)
Erschafft ein neues Textfeld mit der übergebenen Anzahl Spalten.

+ TextEditField (*String*)
Erschafft ein neues Textfeld mit dem übergebenen Text.

+ TextEditField (*String, int*)
Erschafft ein neues Textfeld mit übergebenem Text und Spaltenzahl.

Methoden

+ wasEdited () → *boolean*

Gibt an, ob dieses Textfeld vom Benutzer geändert wurde.

+ commit () → *void*
  Setzt den momentanen Text als Originaltext. Das Textfeld ist damit nicht mehr editiert.
+ rollback () → *void*
  Setzt den Text auf den Originaltext zurück.
+ setText (*String*) → *void*
  überschreibt die Methode der Klasse *JTextComponent*.
  Setzt den Übergebenen Text sowohl als aktuellen als auch Originaltext. Das Textfeld ist damit nicht mehr editiert.

D.16.23 Klasse Triplet

extends *java.lang.Object*
Diese Klasse hält ein geordnetes Tiplet von Instanzen beliebiger Klassen. Der Zugriff erfolgt direkt auf die Instanzvariablen.

Attribute

+ first:*Object* – das erste Objekt
+ second:*Object* – das zweite Objekt
+ third:*Object* – das dritte Objekt

Konstruktoren

+ Triplet (*Object, Object, Object*)
  Erschafft ein neues Triplet mit den übergebenen Objekten.

D.16.24 Klasse UndoStack

extends *java.util.Stack*
Diese Klasse erweitert die Struktur Stack um ein buffer Element

Konstruktoren

+ UndoStack ()
  Erstellt einen neuen Stack unbegrenzter Kapazität
+ UndoStack (*int*)
  Erstellt einen neuen Stack mit maximaler Kapazität maxSize

Methoden

+ push (*Object*) → *Object*
  überschreibt die Methode der Klasse *Stack*.
  Fügt ein Objekt in den Buffer
+ clearBuffer () → *void*
  Löscht den Buffer
+ isBufferEmpty () → *boolean*
  Prüft ob der Buffer Empty ist
+ setMaxSize (*int*) → *void*
  Setzt die maximale Stackgröße
– trimSize () → *void*
  Setzt die ggf. die Größe
+ commit () → *void*
  Fügt den Buffer in den Stack

D.16.25 Klasse VecArithmetik

extends *java.lang.Object*
Diese Klasse stellt einige Funktionen zum Rechnen mit Vektoren zur Verfügung

Methoden

+$ sub (*Point*, *Point*) → *Point*
Subtrahieren

+$ add (*Point*, *Point*) → *Point*
Addieren

+$ add (*Dimension.Double*, *double*) → *Dimension.Double*
Addieren

+$ add (*Dimension.Double*, *Dimension*) → *Dimension.Double*
Addieren

+$ add (*Dimension.Double*, *Dimension.Double*) → *Dimension.Double*
Addieren

+$ closerThan (*Point2D.Double*, *Point2D.Double*, *double*) → *boolean*
Gibt zurück, ob der Euklidische Abstand der beiden Punkte weniger ist als epsilon.

+$ closerThan (*Point2D.Double*, *double*, *double*, *double*) → *boolean*
Gibt zurück, ob der Euklidsche Abstand zwischen Punkt und Koordinaten weniger ist als epsilon.

+$ putOntoCircle (*Point2D.Double*, *Point2D.Double*, *double*) → *Point2D.Double*
Gibt den Punkt auf dem Kreis zurück, der dem übergebenen Punkt am nächsten ist.
Parameter:

- loc: Punkt, der verlegt werden soll
- center: Mittelpunkt des Kreises
- rad: Radius des Kreises

+$ intersectBeam (*Point2D.Double*, *Point2D.Double*, *double*, *double*, *double*, *double*) → *Point2D.Double*
Rückgabewert: Schnittpunkt des Strahls mit der Strecke oder `Null`, falls sie sich nicht schneiden
Schneidet einen Strahl, der durch einen Anfangspunkt und einen weiteren Punkt gegeben ist, durch den er verlaufen soll, mit einer Strecke, die durch die Koordinaten ihres Anfangs- bzw. Endpunktes gegeben ist.
Parameter:

- center: Startpunkt des Strahls
- aim: Punkt, durch den der Strahl verläuft

D.16.26 Klasse ZoomGraphics

extends *java.lang.Object*
Diese Klasse ermöglicht das einfache Zeichnen bestimmter grafischer Objekte mit einer eingestellten Vergrößerung. Die `protected` Methode mit dem Zusatz `Here` erwarten Punkte, bei denen diese Vergrößerung bereits eingerechnet ist.

Konstruktoren

+ ZoomGraphics (*Graphics*, *double*)
Erschafft eine neue ZoomGraphics, die mit dem angegebenen Zoom in die übergebene Klasse Graphics zeichnet.

Methoden

+ getZoom () → *double*

Gibt die eingestellte Vergrößerung zurück.

\# scale (*double*) → *int*

Rechnet den übergebenen Wert so um, dass er zur Vergrößerung passt.

\# drawStringHere (*String, int, int*) → *void*

Zeicnet einen String an die übergebene Stelle.

\# drawLineHere (*int, int, int, int*) → *void*

Zeichnet eine Linie zwischen den übergebenen Punkten.

\# drawRectHere (*int, int, int, int*) → *void*

Zeichnet ein Rechteckt mit der angegebenen Position und Größe.

\# fillRectHere (*int, int, int, int*) → *void*

Zeichnet ein ausgefülltes Rechteck mit der angegebenen Position und Größe.

\# drawRoundRectHere (*int, int, int, int, int, int*) → *void*

Zeichnet ein Rechteck mit abgerundeten Ecke mit der angegebenen Position und Größe.

\# fillRoundRectHere (*int, int, int, int, int, int*) → *void*

Zeichnet ein ausgefülltes Rechteck mit abgerundeten Ecken mit der angegebenen Position und Größe.

\# drawOvalHere (*int, int, int, int*) → *void*

Zeichnet ein Oval mit der angegebenen Position und Größe.

\# fillOvalHere (*int, int, int, int*) → *void*

Zeichnet ein ausgefülltes Oval mit der angegebenen Position und Größe.

\# fillPolygon (*int[], int[], int*) → *void*

Zeichnet ein ausgefülltes Polygon mit den angegebenen Eckpunkten.

\+ setColor (*Color*) → *void*

Setzt die Zeichenfarbe.

\+ getColor () → *Color*

Gibt die Zeichenfarbe zurück.

\+ drawString (*String, Point2D.Double, Point*) → *void*

Zeichnet einen String an die übergebene Position mit dem übergebenen Offset. Die Vergrößerung wird noch beachtet.

\+ drawString (*String, Point2D.Double*) → *void*

Zeichnet einen String an die übergebene Position. Die Vergrößerung wird noch beachtet.

\+ drawOval (*Point2D.Double, Dimension.Double*) → *void*

Zeichnet ein Oval mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

\+ fillOval (*Point2D.Double, Dimension.Double*) → *void*

Zeichnet ein gefülltes Oval mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

\+ drawActiveRect (*Point2D.Double, Dimension.Double*) → *void*

Zeichnet ein als aktiv markiertes abgerundetes Rechteck mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

\+ fillActiveRect (*Point2D.Double, Dimension.Double*) → *void*

Zeichnet ein als aktiv markiertes, ausgefülltes, abgerundetes Rechteck mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

\+ drawMarkedRect (*Point2D.Double, Dimension.Double*) → *void*

Zeichnet ein markiertes abgerundetes Rechteck mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

\+ drawDottedLine (*double, double, double, double*) → *void*

Zeichnet senkrechte oder waagrechte Linien gestrichelt. Schräge Linien werden durchgezogen gezeichnet. Die Vergrößerung wird noch beachtet.

+ drawLine (*double, double, double, double*) → *void*

  Zeichnet eine Linie zwischen den angegebenen Punkten. Die Vergrößerung wird noch beachtet.

+ drawRect (*double, double, double, double*) → *void*

  Zeichnet ein Rechteck mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ drawRect (*Point2D.Double, Dimension.Double*) → *void*

  Zeichnet ein Rechteck mit der angegebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ drawHook (*Point2D.Double*) → *void*

  Zeichnet einen Hook an die angegebene Position. Die Vergrößerung wird noch beachtet.

+ fillRoundRect (*Point2D.Double, Dimension.Double*) → *void*

  Zeichnet ein ausgefülltes Rechteck mit abgerundeten Ecken mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ fillRoundRect (*double, double, double, double*) → *void*

  Zeichnet ein ausgeülltes abgerundetes Rechteck mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ drawRoundRect (*Point2D.Double, Dimension.Double*) → *void*

  Zeichnet ein Rechteck mit abgerundeten Ecken mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ drawRoundRect (*double, double, double, double*) → *void*

  Zeichnet ein Rechteck mit abgerundeten Ecken mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ fillRect (*double, double, double, double*) → *void*

  Zeichnet ein ausgefülltes Rechteck mit der übergebenen Position und Größe. Die Vergrößerung wird noch beachtet.

+ drawSchranke (*Point2D.Double, Dimension.Double*) → *void*

  Zeichnet eine Abschlusslinie an die übergebene Position, die senkrecht zur übergebenen Richtung ist. Die Vergrößerung wird noch beachtet.

+ drawArrowHead (*Point2D.Double, Dimension.Double*) → *void*

  Zeichnet eine Pfeilspitze an die übergebene Position in der übergebenen Richtung. Die Vergrößerung wird noch beachtet.

D.17 Package score.Verifikator

D.17.1 Klassenhierarchie

· *java.lang.Object*
  ↪ *score.Verifikator.VProperties*
  ↪ *score.Verifikator.VPropertyWindow*
  ↪ *score.Verifikator.VWindow*
  ↪ *score.Verifikator.VWindow.VerifMesg*
  ↪ *score.Verifikator.Verifikator*

D.17.2 Klasse Verifikator

extends *java.lang.Object*
Der Verifikator führt die Überprüfungen des Statecharts durch. Die einzelnen Routinen sind in privaten Methoden abgelegt. Welche Tests ausgeführt werden, wird aus den *VProperties*, die beim Erschaffen übergeben werden, ausgelesen.

Konstruktoren

+ Verifikator (*VProperties, GrafSCModell, LogDModell*)

  Erschafft einen neuen Verifikator und beginnt die Überprüfung mit der in den übergebenen VProperties gesetzten Auswahl.

Methoden

+$ wasOKlastTime (*GrafSCModell, LogDModell*) $\rightarrow$ *boolean*

  Gibt zurück, ob das übergebene Grafische Modell in Ordnung war. Wenn es sich seit dem letzten Test nicht geändert hat, wird kein erneuter Test durchgeführt.

D.17.3 Klasse VProperties

extends *java.lang.Object*
In dieser Klasse werden die Einstellungen für einen Testlauf gespeichert.

Attribute

+ option:*boolean* – voreingestellte Auswahl der Test-Optionen
+ lab:*String* CONST – Namen der Text-Optionen
+$ KEEP_GOING:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_DEF_ENTRIES:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_HISTORY:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_HOOKS_IN_NON_SUPER:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_CROSS_PART_BORDERS:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_PARTITIONS:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_ARROW_STATES:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_ALL_EVENTS_USED:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_ALL_VARIABLES_USED:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_N1_ARROWS_START:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_TRIGGER_HIERARCHY:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_TRIGGERS_THROWN:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_CONNECTOR_LOOPS:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_ACCESSIBLE:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_END_STATES:*int* CONST – Kennzeichner der einzelnen Test-Optionen
+$ DO_LABELS:*int* CONST – Kennzeichner der einzelnen Test-Optionen

D.17.4 Klasse VPropertyWindow

extends *java.lang.Object*
implements *javax.swing.event.ChangeListener,*
java.awt.event.ActionListener

Diese Klasse stellt die Eigenschaften die vom Verifikator geprüft werden können dar

Konstruktoren

+ VPropertyWindow (*VProperties*)

  Konstruktor

D.17.5 Klasse VWindow

extends *java.lang.Object*
implements *java.awt.event.ActionListener,*
javax.swing.event.ListSelectionListener

innere Klassen: *VWindow.VerifMesg*
Stellt das Fenster für die Verifikatorausgabe dar

Konstruktoren

+ VWindow (*GrafSCModell*)
  Konstruktor

Methoden

+ addEntry (*String, MarkableTyp*) → *void*
  Einen neuen Eintrag ins Verifikatorlog
+ addEntry (*String, Vector*) → *void*
  Einen neuen Eintrag ins Verifikatorlog
+ addTitle (*String, MarkableTyp*) → *void*
  Einen Titeleintrag schreiben
+ actionPerformed (*ActionEvent*) → *void*
  Auf Dismiss reagieren (schliessen)
+ valueChanged (*ListSelectionEvent*) → *void*
  Auf Selektion in der Liste reagieren (markieren)

D.17.6 Klasse VWindow.VerifMesg

extends *java.lang.Object*
Diese innere Klasse enthält einen String und entweder einen einzelnev MarkableTyp oder einen Vector von solchen. Die Methode toString gibt nur den Text aus, so dass dieser in Listen verwendet werden kann.

Konstruktoren

+ VWindow.VerifMesg (*VWindow, String, MarkableTyp*)
  Erzeugt eine neue VerifMesg mit dem übergebenen Text und einem einzelnen MarkableTyp.
+ VWindow.VerifMesg (*VWindow, String, Vector*)
  Erzeugt eine neue VerifMesg mit dem übergebenen Text und dem Vector von MarkableTypen.

Methoden

+ isSingle () → *boolean*
  Gibt an, ob diese VerifMesg einen einzelnen MarkableTyp enthält.
+ toString () → *String*
  überschreibt die Methode der Klasse *Object*.
  Gibt den Text dieser VerifMesg zurück. Durch diese Methode kann die VerifMesg ordentlich in Listen ausgegeben werden.
+ getMarkableTyp () → *MarkableTyp*
  Gibt den MarkableTyp zurück, falls diese VerifMesg einen einzelnen solchen enthält, ansonsten `Null`.
+ getMarkableTypes () → *Vector*
  Gibt den Vector von MarkableTypen zurück, falls diese VerifMesg einen solchen enthält, ansonsten `Null`.

[done in 81132 ms]

Divide et impera![1]
(Cäsar)

Anhang E

Aufteilung dieser Arbeit

Diese Arbeit wurde als gemeinschaftliche Diplomarbeit von Eric Schellhammer und Benjamin Hargarten vorgelegt. Hierbei wurde folgende Aufteilung vorgenommen:

Von Eric Schellhammer wurden die folgenden Kapitel bzw. Programmteile erstellt:

- Kapitel 2. Varianten von Statecharts in der Literatur
- Kapitel 3. Werkzeuge für Statecharts (bis Abschnitt 3.3)
- Die logischen, grafischen und Simulationsmodelle
- Die Klassen für Aktionen, Bedingungen und Ereignisse
- Der Verifikator

Von Benjamin Hargarten wurden die folgenden Kapitel bzw. Programmteile erstellt:

- Kapitel 1. Definition von Statecharts
- Kapitel 3. Werkzeuge für Statecharts (ab Abschnitt 3.3)
- Die grafische Benutzeroberfläche, sowie die Scanner-/Parser-Module
- Das Dialogmodell
- Der Simulator

Die Programmteile wurden von den jeweiligen Autoren beschrieben und dokumentiert. Sämtliche übrigen Kapitel und Programmteile wurden gemeinschaftlich erarbeitet.

[1] Teile und herrsche!

Bilde, Künstler, rede nicht!
(Goethe)

Abbildungsverzeichnis

1.1 Beispiel-Statechart zur Erläuterung von σ und π . . . 23
1.2 Atomarer Zustand mit Identifikation A . . . 27
1.3 Zusammengesetzter Zustand (Typ OR) . . . 27
1.4 Zusammengesetzter Zustand mit Partitionierung (Typ AND) . . . 28
1.5 Ein einfacher Übergang von A nach B . . . 28
1.6 Fehlerhaftes Statechart, da Partitionsgrenzen geschnitten werden. . . . 29
1.7 Ein komplexer Übergang . . . 29
1.8 Korrekte Defaulteinstiege . . . 30
1.9 Fehlerhafte Defaulteinstiege . . . 30
1.10 Korrekte Historyeinstiege . . . 30
1.11 Fehlerhafte Historyeinstiege . . . 31

2.1 überlappende Zustände . . . 41
2.2 Auflösung der überlappenden Zustände . . . 41
2.3 unsichere Zustände . . . 42
2.4 Übergänge mit mehr als einem End- bzw. Startzustand . . . 43
2.5 Reihenfolge von Aktionen bei statischen Verzweigungen . . . 44
2.6 Produzent und Konsument sind über einen Synch-Zustand verbunden 45
2.7 äquivalente zusammengesetzte Übergänge . . . 47
2.8 Defaulteinstieg mit Bedingung . . . 48
2.9 Unterschiede bei Simulation und Quelltext . . . 48
2.10 Prioritäten an Übergängen . . . 49
2.11 Strukturelle Priorität . . . 50

3.1 Darstellung von Pfeilen bei verschiedenen Tiefenvergrößerungen . . . 59
3.2 Partieller Tiefenvergößerung bei den Zuständen A und H bzw. E . . . 60
3.3 Einfügen eines 2-1 Pfeiles . . . 62
3.4 Radiobuttons im Statechart . . . 67
3.5 Ein zunächst ungültiger Pfeil wird korrigiert . . . 71
3.6 Konflikte bei identischen Ereignissen . . . 73
3.7 alle Zustände können verlassen werden . . . 74
3.8 Notwendigkeit eines Defaulteinstiegs . . . 75
3.9 ein Schreiben/Schreiben-Race . . . 76
3.10 Optimierung des grafischen Layouts . . . 84
3.11 Optimierung nach Anzahl der Zustände . . . 84
3.12 Optimierung durch Parallelisierung . . . 86

3.13 Optimierung nach Anzahl der Variablen 87

4.1 Konstruktion mit Bedingungen am Defaulteinstieg 94
4.2 Probleme bei lokalem Namensraum für Zustände 95
4.3 Inkonsistenz bei lokalem Namensraum für Variablen 96
4.4 Beispiel zu Aktionen und Ereignissen 98
4.5 Transition mit mehreren Ereignissen 99
4.6 Analoge Transition mit nur einem Ereignis 99
4.7 Analoge Transition mit Konnektor 99

5.1 Inspektor bei Start des Programms 106
5.2 Grafische Ansicht 107
5.3 Inspektor mit markiertem AND-Zustand 108
5.4 Inspektor mit markiertem Pfeil 108
5.5 Inspektor mit markiertem Textlabel 109
5.6 Inspektor mit markiertem Button 109
5.7 Verschiedene Zustände in der Ansicht 113
5.8 Verschiedene Transitionen in der Ansicht 114
5.9 Defaulteinstieg, Konnektor, Historyeinstieg und tiefer Historyeinstieg in der Ansicht 115
5.10 Die Werkzeugleiste 115
5.11 Die Ersetzungstabelle 117
5.12 Die Eigenschaftsleiste 118
5.13 Der Dialog-Editor 118
5.14 Die Ereignisliste 121
5.15 Die Aktionsliste 122
5.16 Das Aktionen-Eigenschaftsfenster 123
5.17 Das Simulationsfenster 124
5.18 Ansicht während einer Simulation 124
5.19 Simulationsfenster während einer Simulation 125
5.20 Der Verifikator nach einem Durchlauf 128
5.21 Das Auswahlfenster für den Verifikator 129
5.22 Ein ungültig platzierter Hook 131

6.1 Überblick über die Pakete 136
6.2 Klassenüberblick des Pakets *LogSCM* 137
6.3 Klassenüberblick des Pakets *GrafSCM* 139
6.4 Klassenüberblick des Pakets *SimSCM* 141
6.5 Aufbau des logischen Dialogmodells 142
6.6 Überblick über die Klassen des Statechart-Editors 143
6.7 Hierarchie der Simulation 145
6.8 Übersicht des Simulationsablaufs 147
6.9 Ablauf der Aktivierung im Simulationsmodell 149
6.10 Hierarchie des Quelltextexports 150

A.1 Die Teemaschine, die simuliert wird 156
A.2 Die Dialoge der Teemaschine 157
A.3 Die Gesamtansicht der Teemaschine 158
A.4 Die Partition für den Ein/Ausschalter 160

A.5 Die Partition für die Stoppuhr 161
A.6 Die Partition für den Wassertank 161
A.7 Die Partition, die den Tee im Ziehbehälter beschreibt 162
A.8 Die Partition, die das Wasser im Ziehbehälter beschreibt 163
A.9 Die Partition für die Teekanne 163
A.10 Der Simulationsdialog des Anrufbeantworters 165
A.11 Die Gesamtansicht des Anrufbeantworters 166
A.12 Verhalten bei einem ankommenden Anruf 169
A.13 Das Abspielen von Anrufen 170

Der Brunnen der
Vergangenheit ist tief.
(Thomas Mann)

Literaturverzeichnis

[Abd93] Abdelhamid, Rames: *Das Vieweg LaTeX-Buch.* Vieweg, 2. Aufl., 1993

[Alh98] Alhir, Sinan Si: *UML in a nutshell.* O'Reilly, 1998

[Bal99] Balzert, Heide: *Lehrbuch der Objektmodellierung: Analyse und Entwurf.* Spektrum, 1999

[Beek94] von der Beek, M: *A comparison of statechart variants.* in *Formal Techniques in Real-Time and Fault-Tolerant Systems.* L. de Roever and J. Vytopil, Eds. Lecture Notes in Computer Science, vol. 863. Springer-Verlag, New York, 128-148.

[Brö00] Brössler, Peter und Siedersleben, Johannes (Hrsg.): *Softwaretechnik – Praxiswissen für Software-Ingenieure.* Hanser, 2000

[Brü93] Brück, R.: *Entwurfswerkzeuge für VLSI-Layout.* Carl Hanser Verlag, 1993

[Cam98] Cameron, Debra: *GNU Emacs – kurz & gut.* O'Reilly, 1998

[Ced93] Cederqvist, Per: *Version Management with CVS.* Signum Support, 1993

[Dal97] Dalheimer, Matthias Kalle: *Linux – Wegweiser zur Programmierung & Entwicklung.* O'Reilly, 1997

[Dud96] *Bd. 1. Duden, Rechtschreibung der deutschen Sprache.* 21. Aufl. Dudenverlag, 1996

[Goo94] Goosens, Michel / Mittelbach, Frank / Samarin, Alexander: *Der LaTeX-Begleiter* Addison-Wesley, 1994

[Har87b] Harel, David: *Statecharts: a Visual Formalism for Complex Systems.* Science of Computer Programming, 8, 3, pp. 231-274, 1987

[Har88] Harel, David: *On visual formalism.* Communications of the ACM, Vol. 31, No. 5, pp. 514-531, May 1988

[HK99a] Hitz, Martin / Kappel, Gerti: *UML@work – Von der Analyse zur Realisierung.* dpunkt-Verlag, 1999

[HN96] Harel, David / Naamad, A.: *The Statemate Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology, pp. 293-333, 1996

[HP98b] Harel, David / Politi, Michal: *Modeling Reactive Systems with Statecharts.* McGraw-Hill, 1998

[Hor99a] Horrocks, Ian: *Constructing the User Interfaces with Statecharts.* Addison-Wesley, 1999

[Java1.2] *Java 2 Platform, Standard Edition, v1.2.2 API Specification* http://java.sun.com/products/jdk/1.2/docs/api/index.html, Sun Microsystems Inc., 2000

[Lat90] *Grund- und Aufbauwortschatz Latein.* 1. Aufl., Klett, 1999

[Len90] Lengauer, T.: *Combinatorical Algorithms for Integrated Circuit Layout.* Wiley/Teubner, 1990

[Lev95] Leveson, N. G. et. al.: *Requirements specification for process-control systems.* IEEE Trans. Softw. Eng. 20, pp. 684-707, 1994

[Mac] Mackensen, Lutz: *Großes Handbuch der Zitate und Redensarten.* Buch und Zeit

[Nickel] Nickel, Ulrich A.: *Einsatz von UML/Statecharts$^+$ und SDL$^+$ Spezifikation von Steuerungssoftware für Fertigungssysteme* http://www.uni-paderborn.de/fachbereich/AG/schaefer/ag_dt/ISILEIT/Magdeburg.zip

[Oest99] Oesterreich, Bernd (Hrsg.): *Erfolgreich mit Objektorientierung: Vorgehensmodelle für die objektorientierte Softwareentwicklung.* Oldenbourg Wissenschaftsverlag, 1999

[Ott90] Ottmann, Thomas und Widmayer, Peter: *Algorithmen und Datenstrukturen.* BI-Wissenschafts-Verlag, 1990

[Ora91] Oram, Andrew und Talbott, Steve: *Managing Projects with make.* O'Reilly & Associates, 1991

[Pnu91] Pnueli, A. / Shalev, M.: *What is in a step: On the semantics of statecharts.* In *Proceedings of the Symposium on Theoretical Aspects of Computer Software.* Lecture Notes in Computer Science, vol. 526. Springer-Verlag, pp. 244-264.

[Pratchett] Pratchett, Terry: *Interesting Times.* Corgi, 1997

[Pra97] Pratt, Terrence und Zelkowitz, Marvin: *Programmiersprachen – Design und Implementierung.* Prentice Hall, 1997

[Rum99] Rumbaugh J. / Jacobson, I. / Booch, G.: *The Unified Modelling Language Reference Manual.* Addison-Wesley, 1999

[Sch93] Schöning, Uwe: *Theoretische Informatik kurz gefasst.* BI Wissenschaftsverlag, 1992

[Schu99] Schubert, Stephan: *Konsistenz und Werkzeugunterstützung von Systemspezifikationen in UML*. Diplomarbeit Mathematik mit Studienrichtung Informatik, Universiät Hannover, 1999

[Sed92] Sedgewick, Robert: *Algorithmen in C.* Addison-Wesley, 1992

[Tut12] *The Java Tutorial* http://java.sun.com/docs/books/tutorial/index.html, Sun Microsystems Inc., 2000

[Wir95] Wirth, Niklaus: *Algorithmen und Datenstrukturen.* 4. Aufl., Teubner, 1995

[Wir96] Wirth, Niklaus: *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996

Heureka!
(Archimedes)

Stichwortverzeichnis

Mathematische Symbole

A, 25
E, 24
H, 24
K, 25
L_B, 26
SC, 20
Z, 20
π, 22
σ, 20
d, 24
h, 24

A

Abgrenzungskriterien, 172
Aktion, 19, 25, 54, 122
  History, 54
  im Quelltext, 87
  im Systemdialog, 82
  in *SCed*, 100
  scheduled, 54
  verzögert, 54
Ansichten, 60
Anwendungsbereiche, 172
Aufteilung, 313
Aussagenlogik, 26

B

Bedingung, 19, 26, 53, 120
  Erweiterungen, 53
  im Quelltext, 87
  im Systemdialog, 81
  in *SCed*, 101
  Sprache der, 26
Benutzerhandbuch, 105

D

Datenformat
  Grammatik, 175
Defaulteinstieg, 30, 39
  mit Bedingungen, 93
  Verifikation, 74
Defaulteinstiegsfunktion, 24
Definition
  Formal, 19
  grafisch, 27
  Grundlagen, 17
  temporal, 31
  von Statecharts, 17
Delay, 55
Determinismus, 49
  der Bedingungen, 72
Dialog, 102
  Editor, 66, 118, 144
  System, 80
  Variablen, 69
  Zugriff, 68
  zum Statechart, 65, 67
    Kommunikation, 68
  zur Simulation, 78
Dialogkomponente, 103
  Button, 103
  Checkbox, 103
    Modellierung, 67
  mit mehreren Ereignissen, 69
  Popup, 103
  Radiobutton, 103
    Modellierung, 67
  Sichtbarkeitstabelle, 53, 68
  Textfeld, 103
  Timer, 103
  untypisch, 119
Dialogmodell, 142
Direktheit, 21

E
Editor
Dialog, 66, 144
Statechart, 57, 143
Eigenschaftsleiste, 118
Eingabemaske, 144
Ereignis, 19, 24, 51, 121
aufgeschoben, 52
deferred, 52
im Quelltext, 87
im Systemdialog, 80
in *SCed*, 98
in Bedingungen, 64
negiert, 64
Sichtbarkeitsdauer, 51
Ermittlung
aktivierter Oberzustände, 34
aktivierter Unterzustände, 34
deaktivierter Oberzustände, 35
deaktivierter Unterzustände, 34
des Folgeereignisses, 33
des Folgezustand, 32
Error
InvalidConstructionError, 219
Exception
NoTargetStateException, 284

F
Fixpunktfreiheit, 21
Folgeereignis, 33
Folgezustand, 32

G
Generische Charts, 40, 96
Grafischer Editor, 112
grafisches Statechart-Modell, 138
Grammatik, 175

H
Historyeinstieg, 30
Historyfunktion, 24
Historykonnektor, 45
Hook, 61

I
Inspektor, 62, 106
zu Dialogkomponenten, 67
Interface
Actor, 182
ConditionAskable, 196
DialogAktion, 183
DialogCompManager, 250
DialogCompServer, 250
ErrorPrinter, 296
EventAskable, 216
EventHolder, 183
HistoryAktion, 184
Logger, 203
Observer, 296
UndoManager, 296
VarServer, 290

K
Kante, 25, 28
Klasse
AbstractModell, 179
AbstractParser, 204
Aktion, 184
AktionDialog, 185
AktionEdit, 186
AktionProperty, 186
AktionPropertyChange, 187
AktKomplex, 188
AnimatedImage, 296
BedAND, 196
BedAtomar, 197
Bedingung, 197
BedKomplex, 198
BedNOT, 199
BedOR, 200
BedParser, 204
BlackLabel, 297
BufferedCanvas, 273
CalledAction, 189
ClearHistoryAktion, 189
Commander, 205
CommandPort, 207
CommandWindow, 208
Compare, 297
CompVarBed, 200
DeepClearHistoryAktion, 190
DialogEditor, 213
Dimension, 298
Dimension.Double, 298
DlgButton, 251

DlgCheckbox, 251
DlgKomponente, 252
DlgKompSelektor, 214
DlgPopup, 254
DlgRadiobutton, 255
DlgText, 255
DlgTimer, 256
DrawableRubber, 274
EasyFileWriter, 299
EasyStringWriter, 300
Ereignis, 216
EreignisEdit, 217
EventDialog, 217
ExportAktion, 190
ExportBedingung, 201
ExportVariable, 271
FlatButton, 300
FlatRadioButton, 300
GPAbschnitt, 219
GrafAND, 221
GrafHook, 222
GrafKonnektor, 225
GrafModellSicht, 226
GrafModellSicht.visHook, 231
GrafModellSicht.visText, 231
GrafPfeil, 231
GrafPunkt, 234
GrafSCModell, 239
GrafText, 240
GrafTopLevelZust, 241
GrafTyp, 242
GrafZustand, 243
HasHistoryBed, 201
IncVarAktion, 191
InlineEventDialog, 218
Inspector, 208
InStateBed, 202
Interpreter, 208
Intersect, 301
JavaAktion, 192
JavaBed, 202
KonDefault, 259
KonHistory, 259
Konnektor, 259
KonUniversal, 260
LaTeXDoclet, 215
LaTeXString, 301
LogComponent, 260
LogDModell, 257
LogPfeil, 260
LogPfeil_1_1, 262
LogPfeil_1_N, 263
LogPfeil_N_1, 264
LogPunkt, 265
LogSCModell, 267
MainWindow, 209
MarkableTyp, 181
MetaPostGraphics, 301
NumField, 303
Observable, 303
Painter, 275
Pair, 304
Parser, 210
Partitioner, 247
PropertyView, 211
RubberBand, 275
RubberCircle, 276
RubberFilledCircle, 276
RubberOval, 276
RubberRectangle, 277
RubberRoundRect, 277
RubberSet, 278
RubberText, 279
Scale, 304
Scanner, 212
SCCanvas, 279
SCEditor, 280
SCInterpreter, 212
ScrollCanvas, 283
SCRun, 290
SetDialogOnAktion, 192
SetDialogVisibleAktion, 193
SetVarAktion, 194
SimAND, 284
SimBasic, 285
SimConn, 287
SimHist, 287
SimOR, 288
SimSCModell, 289
Simulation, 292
SimVariable, 293
SimWindow, 293
start, 179
StateChart, 271

StringCheck, 305
SuperList, 249
TextEditField, 305
ThrowAktion, 195
Timer, 272
Token, 213
TranslationWindow, 283
Triplet, 306
UndoStack, 306
Unterpunkt, 249
Variable, 294
VecArithmetik, 307
Verifikator, 309
VProperties, 310
VPropertyWindow, 310
VWindow, 311
VWindow.VerifMesg, 311
ZoomGraphics, 307
ZustAND, 268
ZustandAtom, 268
ZustKomplex, 269
ZustOR, 270
Konnektor, 31, 42, 115
AND, 43
OR, 43
Diagrammkonnektor, 43
History, 45
Ausgänge, 46
deep, 46
in *SCed*, 97
in UML, 44
Schleife, 72
Universal, 97
Kriterien
Abgrenzung, 172
Kann, 172
muss, 171

L

LogDM, 142
logisches Dialogmodell, 142
logisches Statechart-Modell, 137

M

Menge
der Aktionen, 25
der Ereignisse, 24
der Historyeinstiege, 24
der Kanten, 25
der Zustände, 20
Modell
in *SCed*
grafisch, 138
logisch, 137
Quelltext, 141
Simulation, 141

N

Namensraum
in *SCed*, 95

O

Oberzustand, 34, 35
Eindeutigkeit, 20
Optimierer, 83
orthogonale Komponente, 39

P

Parametrisierung, 39
AND, 40
OR, 40
in *SCed*, 94
Partitionsfunktion, 22
Pfeil
ungültiger, 71
Pflichtenheft, 171
Popup, 119
Priorität
strukturell, 50
Zahl, 49

Q

Quelltext
Aktion, 87
Bedingung, 87
Ereignis, 87
Generator, 86
in der Simulation, 79
zum Statechart, 65
Quelltextexport, 126, 149

R

Race, 76
Radiobutton, 67
Rahmenbedingung, 52

Redo, 125, 150
objektbezogen, 63

S

Scope, 50
Simulation
Ablauf, 145
Simulationsablauf, 145
Simulator, 77, 123, 144
Verwaltete Daten, 144
Speichern, 150
Statechart, 20
Editor, 57, 143
Kommunikation mit Dialog, 68
Konfiguration eines, 31
mit Dialog, 65
mit Quelltext, 65
Speicherstruktur, 57
Werkzeuge, 57
zu Dialogkomponenten, 67
Zustand eines, 31
statische Reaktion, 39
Synchronisationsbalken, 44
Systemdialog, 80
Aktion, 82
Bedingung, 81
Ereignis, 80

T

Temporale Auswertung, 31
Testen, 77
Tiefe, 22
Vergrößerung, 58
global, 60
partiell, 59
Time-Out, 55
Timer, 119
Transition, 19, 28, 113

U

Übergang, 19, 28, 46
1-N, 97
ausloserfrei, 50
Erweiterungen, 50
hinzufügen, 61
N-1, 97
triggerless, 50
zusammengesetzt, 47
hinzufügen, 61
Umgebung
Hardware, 173
Software, 173
Undo, 125, 150
objektbezogen, 63
Unterzustand, 34
Partition, 22
transitive Hülle, 21
Fixpunktfreiheit, 21
Unterzustandsfunktion, 20
Symmetriefreiheit, 22

V

Variable, 120
in *SCed*, 101
zu Dialogkomponenten, 69, 103
Verbindung *siehe* Übergang 46
Verifikator, 71, 149

W

Werkzeug, 57
Editor
Dialog, 66
Programmstruktur, 69
Statechart, 57
Optimierer, 83
Quelltextgenerator, 86
Simulator, 77
Verifikator, 71
Werkzeugleiste, 115

Z

Zeichenfläche, 112
Zeit, 102
in der Simulation, 79
Modell, 54
asynchron, 55
kontinuierlich, 54
synchron, 55
Zielbestimmung, 171
Zielgruppe, 172
Zustand, 18, 27, 112
AND, 18, 23, 39
OR, 18, 22, 39
Aktivieren, 147
atomar, 18, 20, 38
Partition, 22

Erreichbarkeit, 73
Erweiterungen, 40
Hierarchie, 39
in *SCed*, 93
nichtatomar
  Partition, 22
parametrisiert, 39
Pseudozustand, 44
Synch, 45
terminal, 38
transient, 38
überschneidend, 40
unsicher, 41